**Patent Office
Canberra**

I, PARAMJIT KOUR, EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. 2002953135 for a patent by SILVERBROOK RESEARCH PTY. LTD. as filed on 02 December 2002.

WITNESS my hand this
Twenty-first day of November 2007

PARAMJIT KOUR
EXAMINATION SUPPORT AND SALES

AUSTRALIA

Patents Act 1990

Provisional Specification

for an invention entitled:
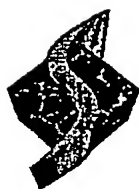
METHOD AND APPARATUS (PEC10)

The invention is described in the following statement: -

SoPEC/MoPEC

Bilithic Printhead Reference

4-4-1-6 version 1.0 draft

October 21, 2002

Document History

Version	Date	Authors	Details
1.0 draft	October 21, 2002	KR	Updated after review at S3
0.1 draft	October 2, 2002	KR	Updated after initial review
0.0 draft	September 30, 2002	KR	Initial draft of document

Contents

INTRODUCTION

1   Background

Silverbrook's bilithic Memjet^{TM} printheads are the target printheads for printing systems which will be controlled by SoPEC and MoPEC devices.

This document presents the format and structure of these printheads, and describes the their possible arrangements in the target systems. It also defines a set of terms used to differentiate between the types of printheads and the systems which use them.

1.1   COMPANION DOCUMENTS

Currently, this document is only concerned with the structure of the printheads and their systems, with regard to the way in which dot data is loaded.

Refer to the Bilithic Printhead Specification [1] for the complete description of the functionality of these devices.

This document relies on certain definitions and details presented in Bilithic Printhead Specification [1].

1.2   READERSHIP

It is intended that this document be used as a reference for engineers involved in the design work on the SoPEC and MoPEC projects.

BILITHIC PRINTHEAD CONFIGURATIONS

2   Definitions

This document presents terminology and definitions used·to describe the bilithic printhead systems. These terms and definitions are as follows:

- Printhead Type - There are 3 parameters which define the type of printhead used in a system:
  - Direction of the data flow through the printhead (clockwise or anti-clockwise, with the printhead shooting ink down onto the page).
  - Location of the left-most dot (upper row or lower row, with respect to V_+).
  - Printhead footprint (type A or type B, characterized by the data pin being on the left or the right of V_+, where V_+ is at the top of the printhead).
- Printhead Arrangement - Even though there are 8 printhead types, each arrangement has to use a specific pairing of printheads, as discussed in Section 3. This gives 4 pairs of printheads. However, because the paper can flow in either direction with respect to the .printheads, there are a total of eight possible arrangements, e.g. Arrangement 1 has a Type 0 printhead on the left with respect to the paper flow, and a Type 1 printhead on the right. Arrangement 2 uses the same printhead pair as Arrangement 1, but the paper flows in the opposite direction.
- Color 0 is always the first color plane encountered by the paper.
- Dot 0 is defined as the nozzle which can print a dot in the left-most side of the page.
- The Even Plane of a color corresponds to the row of nozzles that prints dot 0.

Note that throughout this document, where the various printheads and systems are presented, the printheads **always** shoot ink down onto the page.

Figure 1 shows the 8 different possible printhead types. Type 0 is identical to the Right Printhead presented in Figure 3 in [1], and Type 1 is the same as the Left Printhead as defined in [1].

While the printheads shown in Figure 1 look to be of equal width (having the same number of nozzles) it is important to remember that in a typical system, a pair of unequal sized printheads may be used.
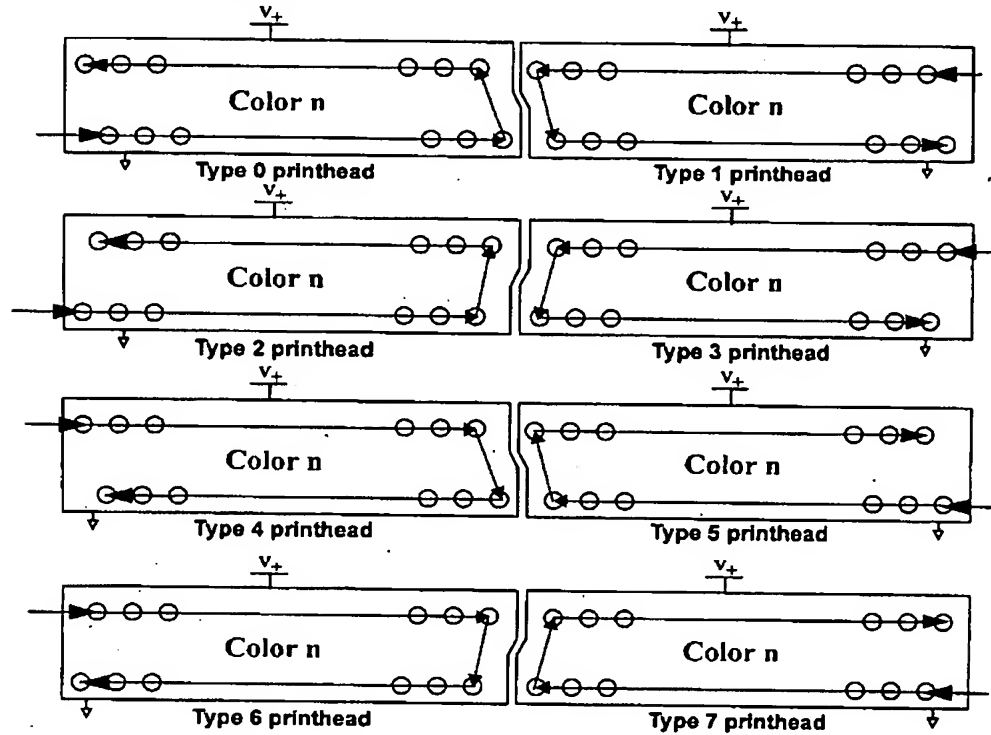

Figure 1. Printhead Types 0 to 7

Table 1 defines the printhead pairing and location of the each printhead type, with respect to the flow of paper, for the 8 possible arrangements.

Table 1. Definition of the different printhead arrangements

Printhead Arrangement	Printhead on left side with respect to the flow of paper	Printhead on right side with respect to the flow of paper
Arrangement 1	Type 0	Type 1
Arrangement 2	Type 1	Type 0
Arrangement 3	Type 2	Type 3
Arrangement 4	Type 3	Type 2
Arrangement 5	Type 4	Type 5
Arrangement 6	Type 5	Type 4
Arrangement 7	Type 6	Type 7
Arrangement 8	Type 7	Type 6

3   Bilithic Printhead Systems

When using the bilithic printheads, the position of the power/gnd bars coupled with the physical footprint of the printheads mean that we must use a specific pairing of printheads together for printing on the same side of an A4 (or wider) page, e.g. we must always use a Type 0 printhead with a Type 1 printhead etc.

While a given printing system can use any one of the eight possible arrangements of printheads, this document only presents two of them, Arrangement 1 and Arrangement 2, for purposes of illustration. These two arrangements are discussed in subsequent sections of this document. However, the other 6 possibilities also need to be considered.

The main difference between the two printhead arrangements discussed in this document is the direction of the paper flow. Because of this, the dot data has to be loaded differently in Arrangement 1 compared to Arrangement 2, in order to render the page correctly.

3.1   EXAMPLE 1: PRINTHEAD ARRANGEMENT 1

Figure 2 shows an Arrangement 1 printing setup, where the bilithic printheads are arranged as follows:

- The Type 0 printhead is on the left with respect to the direction of the paper flow.
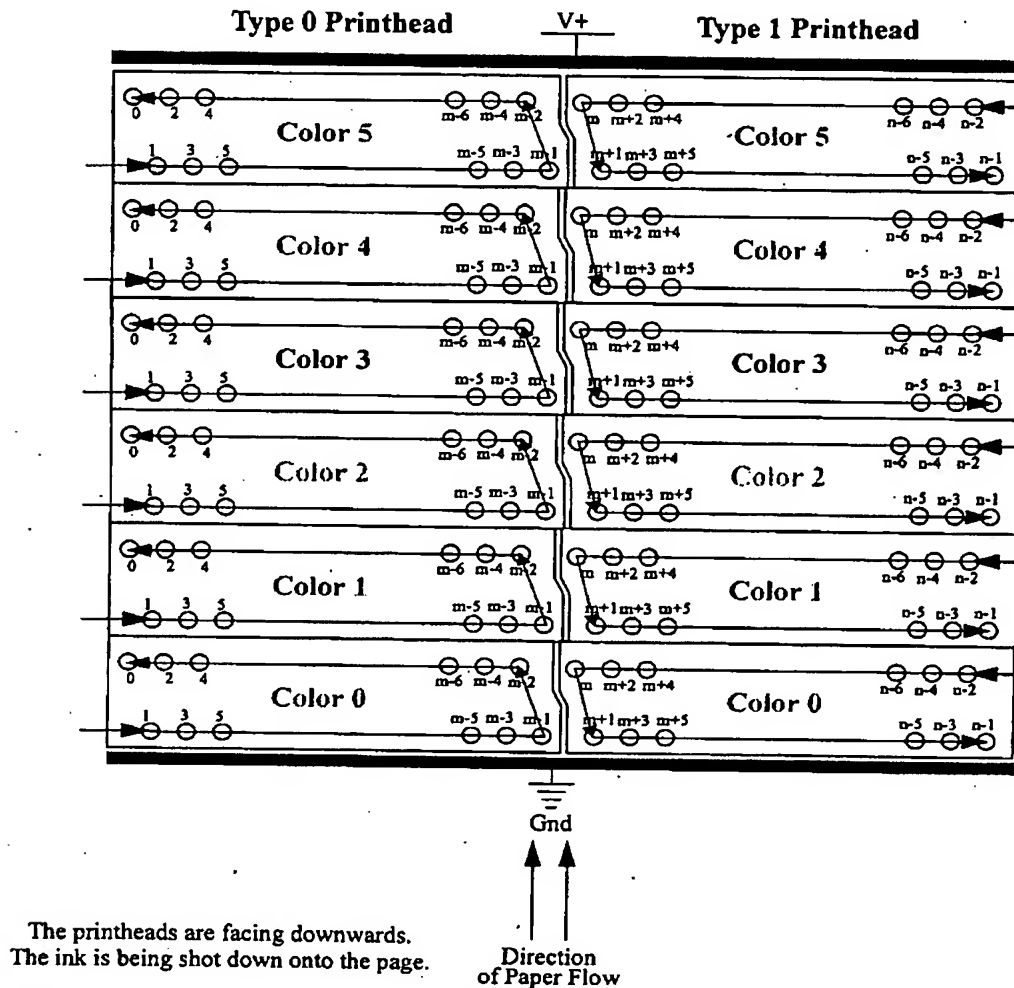- The Type 1 printhead is on the right.

The printheads are facing downwards.
The ink is being shot down onto the page.

Direction
of Paper Flow

Figure 2. Identification of printheads nozzles and shift-register sequences for printheads in Arrangement 1

Table 2 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 2. Order in which the even and odd dots are loaded for printhead Arrangement 1

Dot Sense	Type 0 printhead when on the left	Type 1 printhead when on the right
Odd	Loaded second in descending order.	Loaded first in descending order.
Even	Loaded first in ascending order.	Loaded second in ascending order.

Figure 3 shows how the dot data is demultiplexed within the printheads.

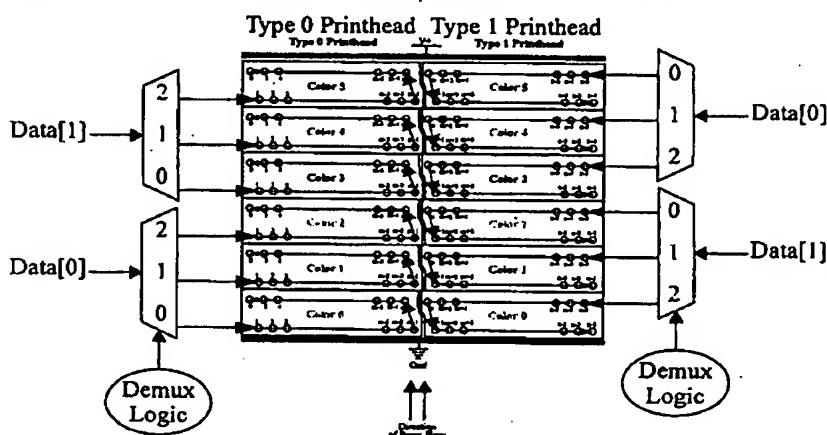**Figure 3. Demultiplexing of data within the printheads in Arrangement 1**

Figure 4 and Figure 5 show the way in which the dot data needs to be loaded into the print-heads in Arrangement 1, to ensure that color 0-dot 0 appears on the left side of the printed page.

Figure 4. Signalling for a Type 0 printhead in Arrangement 1

Figure 5. Signalling for a Type 1 printhead in Arrangement 1

3.2   EXAMPLE 2: PRINTHEAD ARRANGEMENT 2

Figure 6 shows an Arrangement 2 printing setup, where the bilithic printheads are arranged as follows:

- The Type 1 printhead is on the left with respect to the direction of the paper flow.
- The Type 0 printhead is on the right.

The printheads are facing downwards.
The ink is being shot down onto the page.
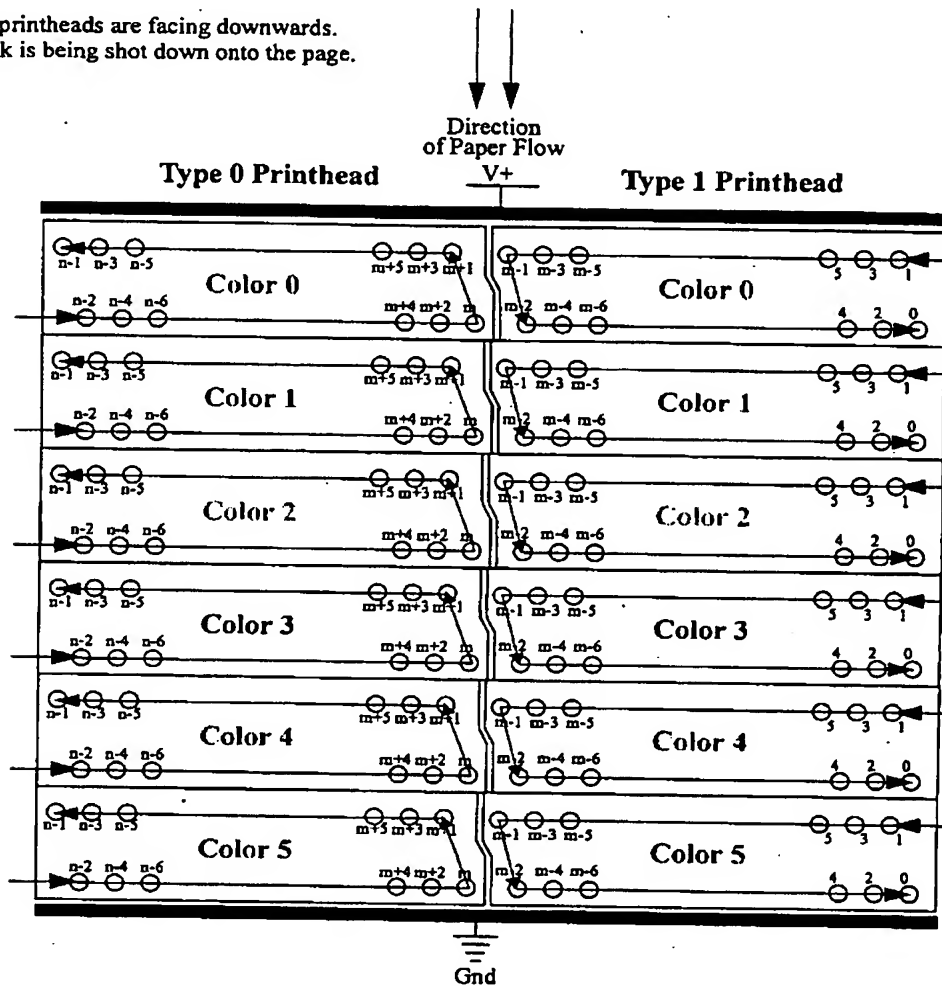
Figure 6. Identification of printheads nozzles and shift-register sequences for printheads in Arrangement 2

Table 3 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 3. Order in which the even and odd dots are loaded for printhead Arrangement 2

Dot Sense	Type 0 printhead when on the right	Type 1 printhead when on the left
Odd	Loaded first in descending order.	Loaded second in descending order.
Even	Loaded second in ascending order.	Loaded first in ascending order.

Figure 7 shows how the dot data is demultiplexed within the printheads.

Figure 7. Demultiplexing of data within the printheads in Arrangement 2

Figure 8 and Figure 9 show the way in which the dot data needs to be loaded into the printheads in Arrangement 2, to ensure that color 0-dot 0 appears on the left side of the printed page.
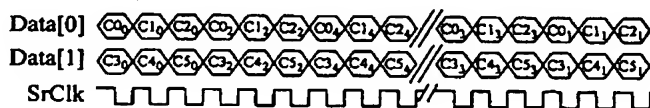

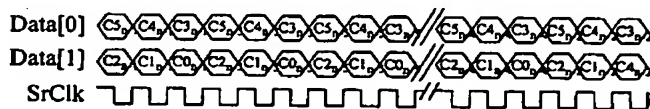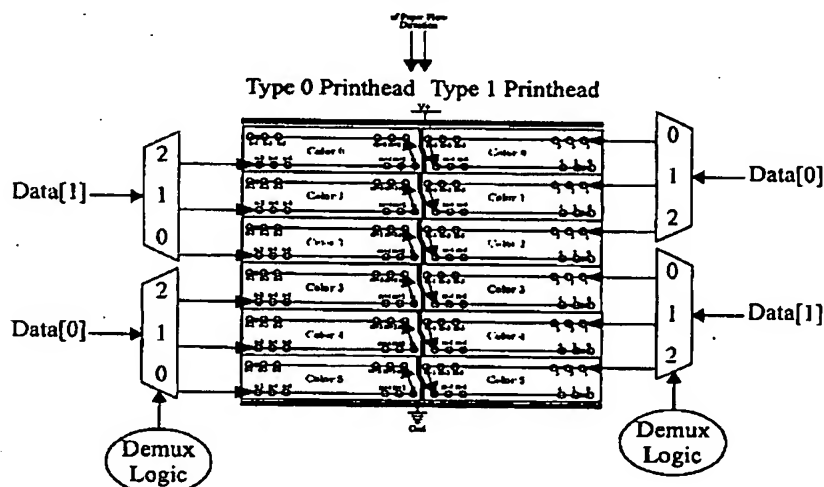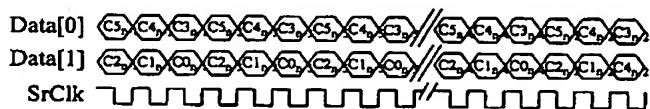**Figure 8. Signalling for a Type 0 printhead in Arrangement 2**
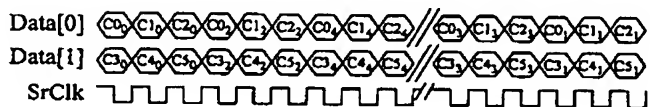
Figure 9. Signalling for a Type 1 printhead in Arrangement 2

3.3   CONCLUSIONS

Comparing the signalling diagrams for Arrangement 1 with those shown for Arrangement 2, it can be seen that the color/dot sequence output for a printhead type in Arrangement 1 is the reverse of the sequence for same printhead in Arrangement 2 in terms of the order in which the color plane data is output, as well as whether even or odd data is output first. However, the order within a color plane remains the same, i.e. odd descending, even ascending.

From Figure 10 and Table 4, it can be seen that the plane which has to be loaded first (i.e. even or odd) depends on the arrangement. Also, the order in which the dots have to be loaded (e.g. even ascending or descending etc.) is dependent on the arrangement.

If the device controlling the printheads can re-order the bits according to the following criteria, then it should be able to operate in all the possible printhead arrangements:
- Be able to output the even or odd plane first.
- Be able to output even and odd planes in either ascending or descending order, independently.
- Be able to reverse the sequence in which the color planes of a single dot are output to the printhead.
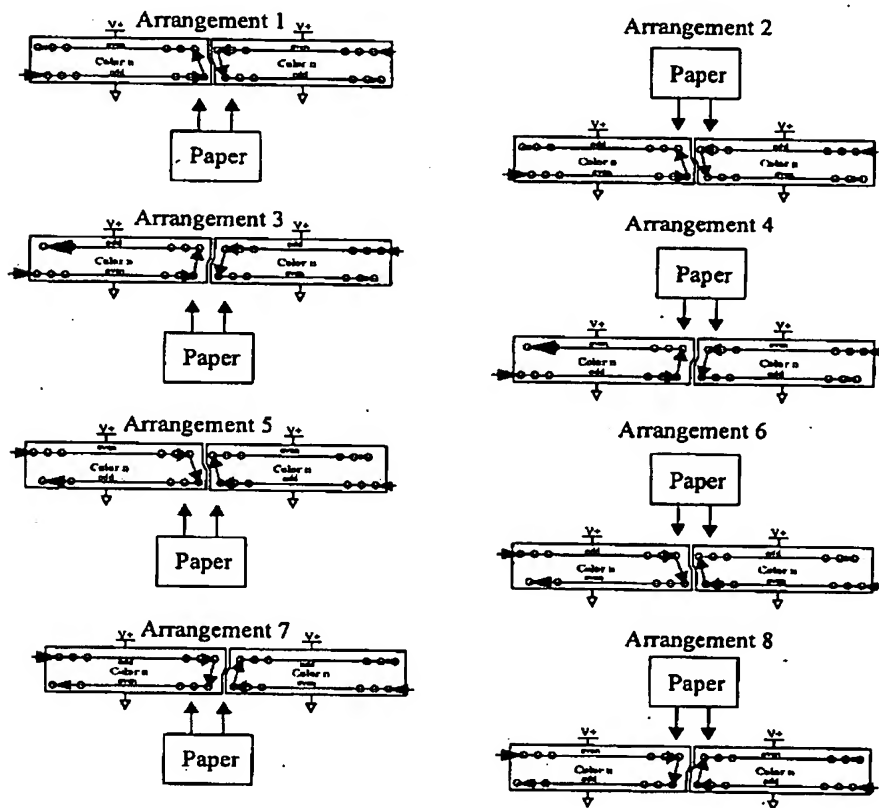
Figure 10. All 8 Printhead Arrangements

Table 4. Order in which even and odd dots and planes are loaded into the various printhead arrangements

Printhead Arrangement	Left side of printed page	Right side of printed page
Arrangement 1	Even ascending loaded first	Odd descending loaded first
	Odd descending loaded second	Even ascending loaded second
Arrangement 2	Odd descending loaded first	Even ascending loaded first
	Even ascending loaded second	Odd descending loaded second
Arrangement 3	Odd ascending loaded first	Even descending loaded first
	Even descending loaded second	Odd ascending loaded second
Arrangement 4	Even descending loaded first	Odd ascending loaded first
	Odd ascending loaded second	Even descending loaded second
Arrangement 5	Odd ascending loaded first	Even descending loaded first
	Even descending loaded second	Odd ascending loaded second
Arrangement 6	Even descending loaded first	Odd ascending loaded first
	Odd ascending loaded second	Even descending loaded second
Arrangement 7	Even ascending loaded first	Odd descending loaded first
	Odd descending loaded second	Even ascending loaded second
Arrangement 8	Odd descending loaded first	Even ascending loaded first
	Even ascending loaded second	Odd descending loaded second

REFERENCES

4   References

[1]   Silverbrook Research: 06-11-02-01-01, *Bilithic Printhead Specification.*

Bi-lithic Printhead Specification

1.0 Basic Requirements

To create a two part printhead, of A4/Letter portrait width to print a page in 2 seconds. Matching Left/Right chips can be of different lengths to make up this length facilitating increased wafer usage. the left and right chips are to be imaged on an 8 inch wafer by "Stitching" reticle images.

The memjet nozzles have a horizontal pitch of 32 um, two rows of nozzles are used for a single colour. These rows have a horizontal offset of 16 um. This gives an effective dot pitch of 16 um, or 62.5 dots per mm, or 1587.5 dots per inch, close enough to market as 1600 dpi.

The first nozzle of the right chip should have a 32 um horizontal offset from the final nozzle of the left chip for the same color row. There is no ink nozzle overlap (of the same colour) scheme employed.

1.1 Power Supply

Vdd/Vpos and Ground supply is made through 30 um wide pads along the length of the chip using conductive adhesive to bus bar beside the chips. Vdd/Vpos is 3.3 Volts. (12V was considered for Vpos but routing of CMOS Vdd at 3.3V would be a problem over the length of the chips, but this will be revisited).

1.2 MEMS cells

The current memjet device requires 180nJ of energy to fire, with a pulse of current for 1 usec. Assuming 95% efficiency, this requires a 55 ohm actuator drawing 57.4 mA during this pulse.

1.2.1 ISSUE!!!

For 1 pages per 2 second, or ~300 mm * 62.5 (dots/mm) / 2 sec ~= 10 kHz or 100 usec per line. With 1 usec fire pulse cycle, every 100th nozzle needs to fire at the same time. (looking ahead) We have 13824 nozzles across the page, so we fire 138 nozzles at a time. That is about 8 Amperes if all nozzle fire.

That is 8 Amperes is for only 1 colour! 16A * 6 colours = 96 A for all colours.

How many colours could print at the same time. CMYK colour space requires on 2 colours at the time are required, to create map any colour (saturated, full bleed background). But the fixative ink us also required, and 12% coverage of InfraRed ink, means 3.12 inks would be a the "best" worst case. Unfortunately, the peak could be all 6 inks, as colours are not aligned to print the same point at the same time.

With a 138 nozzles * 3.12 inks firing at the same time, at 180 nJ each in 1 usec, the memjet nozzle will average (138*3.12*180)/1000 = 78 W running a full speed.

1.2.2 64um unit cell height

This cell would have 4 line spacing between the odd and even dots, and 8 line spacing between adjacent colours.

1.2.3 80 um unit cell height

This cell would have 5 line spacing between the odd and even dots, and 10 line spacing between adjacent colours.

1.3 Versions

1.3.1 6 Colour 1600 dpi with 64 um unit cell

Left and Right Chip. This version will not be prototyped.

1.3.2 6 Colour 1600 dpi with 80 um unit cell

Left and Right Chip.

1.3.3 4 Colour 800 dpi with 80 um unit cell

For camera application. Single nozzle row per colour.

This version will not be prototyped.

1.4 Air Supply

Air must be supplied to the MEMS region through holes in the chip.

2.0 Head Sizes

The combined heads have 13824 nozzles per colour totalling 221.184mm of print area. Enough to provide full bread for A4 (210 mm) and Letter (8.5 inch or 215.9 mm).

TABLE 1. Head Combinations

Left Head		Right Head	
Stitch Parts	Nozzles per Colour	Stitch Parts	Nozzles per Colour
8	11160	2	2664
7	9744	3	4080
6	8328	4	5496
5	6912	5	6912
4	5496	6	8328

TABLE 1. Head Combinations

Left Head		Right Head	
Stitch Parts	Nozzles per Colour	Stitch Parts	Nozzles per Colour
3	4080	7	9744
2	2664	8	11160

Nozzles per Colour is calculate as (("Stitch Parts" -1)*118+104)*12. Nozzles per row is half this value. Most likely the 8:2 head set will not be manufactured. My current wafer layout, manages to avoid this set, without any loses.

3.0 Interface

Each print head has the same I/O signals (but the Left and Right versions might have a different pin out). Pins marked as common can be controlled by the signal from the

TABLE 2. I/O pins

Name	I/O	Function	Common	Max Speed (MHz)
Data[0-1]	I	Dot data for colours 0 - 5, using Differential Signalling (DataL the complementary signal), colours[0-2] on Data[0], colour[3-5] on Data[1]	No	300
	O	Feedback for CMOS testing (*LSyncL*=1, *ReadL*=0) and (*LSyncL*=0, *ReadL*=0) [0] - nozzle test result [1] - temperature		
DataL[0-1]	I	complementary signal of Data[0-1]		
	O	Feedback for CMOS testing (*LSyncL*=1, *ReadL*=0) and (*LSyncL*=0, *ReadL*=0) [0] - nozzle test result [1] - temperature		
SrClk	I	Dot data shift clock using Differential Signalling (SrClkL the complementary signal)	No[a]	600[b]
SrClkL	I	complementary signal of SrClk		
ReadL	I	Data[0-1]/DataL[0-1] in output mode (driving non-differential)	Yes	1
FrClk	I	Fire pattern shift clock	Yes	1
Pr	I	Pulse Profile for all colours	Yes	1[c]
LsyncL	I	0 - Capture dot data for next print line	Yes	0.1[d]

a. Functionally could be common, but for timing/electrical reasons should run point to point.

b. 300 MHz clock, so edges are 600 Mhz rate

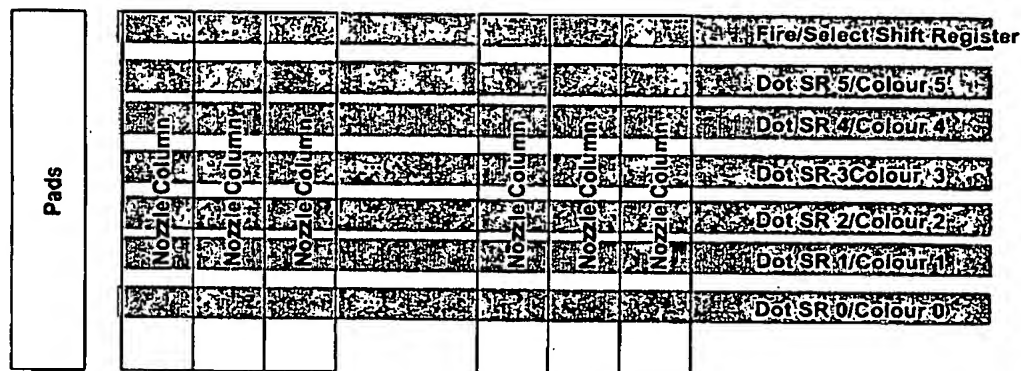c. 1 MHz cycle, but the resolution of the mark/space ratio may require 50 ns.

d. 10 kHz cycle, with minimum low pulse of 10 ns (no maximum).

controller (SOPEC).

3.1 Dot firing

To fire a nozzle, three signals are need. A dot data, a fire signal, and a profile. When all signals are high, the nozzle will fire.

FIGURE 1. Print head structure

The dot data is provide to the chip through a **dot shift register** with input *Data[x]*, and clocked into the chip with *SrClk*. The dot data is multiplex on to the *Data* signals, as *Dot[0-2]* on *Data[0]*, and *Dot[3-5]* on *Data[2]*. After the dots are shifted into the **dot shift register,** this data is transfer into the **dot latch**, with a low pulse in *LsyncL*. The value in the **dot latch** forms the dot data used to fire the nozzle. The use of the **dot latch** allows the next line of data to be loaded into the **dot shift register**, at the same time the dot pattern in the **dot latch** is been fired.

Across the top of a column of nozzles, containing 12 nozzles, 2 of each colour (odd and even dots, 4 or 5 lines apart), is two **fire register** bits and a **select register** bit. The **fire registers** forms the **fire shift register** that runs length of the chip and back again with one register bit in each direction flow.

FIGURE 2. Column Structure
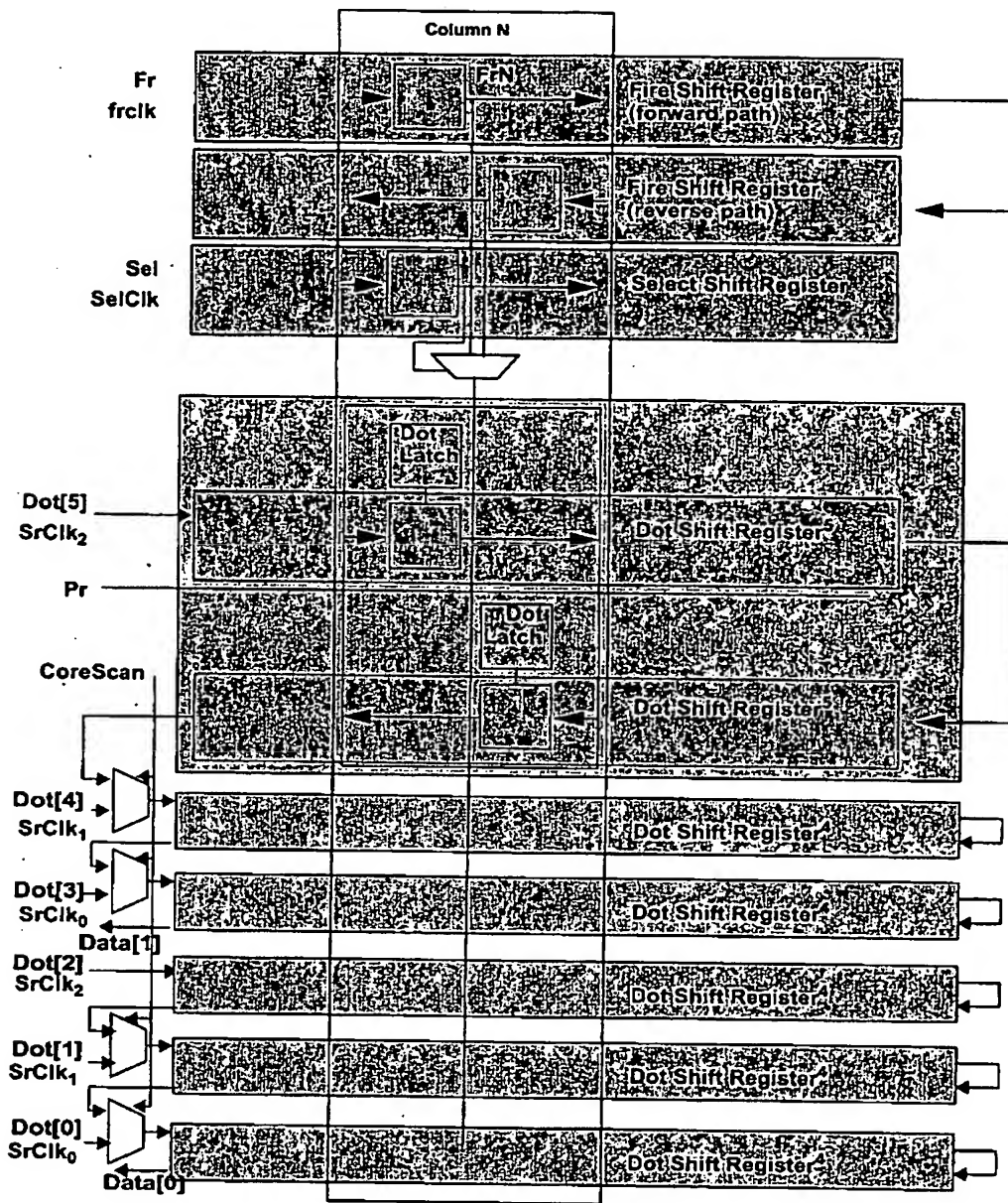
The **select register** forms the **Select Shift Register** that runs the length of the chip. The **select register**, selects which of the two **fire registers** is used to enables this column. A '0' in this register selects the forward direction **fire register**, and a '1' selects the reverse direction **fire register**.

The third signal need, the profile, is provide for all colours with input Pr across the whole colour row at the same time (with a slight propagation delay per column).

3.2 Dot Shift Register Orientation

The left side print head (chip) and the right side print head that form complete bi-lithic print head, have different nozzle arrangement with represent to the dot order mapping of the dot shift register to the dot position on the page.

FIGURE 3. Print head dot shift register dot mapping to page

Section A-A Through Even nozzles
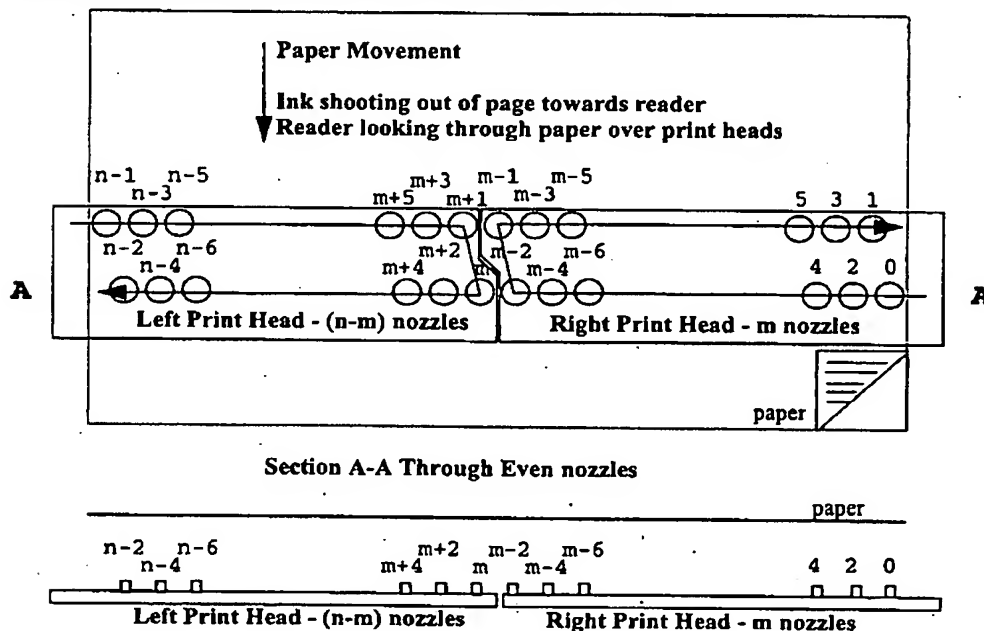
With this mapping, the following data streams will need to provided.

TABLE 3. Head Combinations shift patterns (n=13824)

Left Head			Right Head	
Size	n-m	dot order	m	
7:3	9744	[13822,13820,13818,...,4084,4082,4080,] line y+5 [4081,4083,4085,....,13819,13821,13823] line y	4080	[1,3,5,...,4075,4077,4079,] line y+5 [4078,4076,4074,....,4,2,0] line y
6:4	8328	[13822,13820,13818,...,5500,5498,5496,] line y+5 [5497,5499,5501,....,13819,13821,13823] line y	5496	[1,3,5,...,5491,5493,5495,] line y+5 [5494,5492,5490,....,4,2,0] line y
5:5	6912	[13822,13820,13818,...,6916,6914,6912,] line y+5 [6913,6915,6917,....,13819,13821,13823] line y	6912	[1,3,5,...,6907,6909,6911,] line y+5 [6910,6908,6906,....,4,2,0] line y
4:6	5496	[13822,13820,13818,...,8332,8330,8328,] line y+5 [8329,8331,8333,....,13819,13821,13823] line y	8328	[1,3,5,...,8323,8325,8327,] line y+5 [8326,8324,8322,....,4,2,0] line y
3:7	4080	[13822,13820,13818,...,9748,9746,9744,] line y+5 [9745,97447,9749,....,13819,13821,13823] line y	9744	[1,3,5,...,9739,9741,9743,] line y+5 9742,9740,9738,....,4,2,0] line y

The data needs to be multiplex onto the data pins, such that Data[0] has {(C0, C1, C2), (C0, C1, C2)....} in the above order, and Data[1] has {(C3, C4, C5), (C3, C4, C5)....}.

Figure 4 shows the timing of data transfer during normal printing mode. Note *SrClk* has a default state of high. If there are L nozzles per colour, *SrClk* would have $3L$ pulses (and $3L+1$ rising edges).

FIGURE 4. Data Timing During Printing
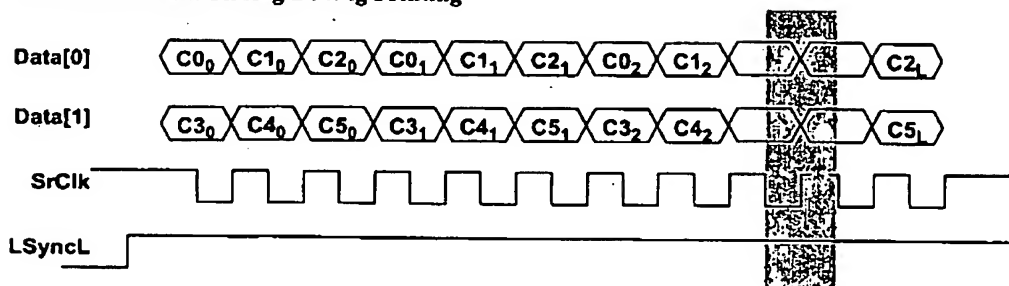
Data requires a setup and hold about the falling edge of *SrClk*. *SrClk* default state is high (needs to return to high after the last data of the line). *LSyncL* rising requires setup to the first rising SrClk, and must stay high during the data transfer.

3.3 Fire Shift Register

The fire shift register controls the rate of nozzle fire. If the register is full of '1's then the you could print the entire print head in a single *FrClk* cycle. *You do not want to do that (4800A)!*

Ideally, a '1' is shifted in to the fire shift register, in every n^{th} position, and a '0' in all other position. In this manner, after n cycle of *FrClk*, the entire print head will be printed.

The **fire shift register** and **select shift registers** allow the generation of a horizontal print line that on close inspection would not have a discontinuity of a "saw tooth" pattern, Figure 5 a) & b) but a "sharks tooth" pattern of c).

FIGURE 5. Print quality

a) Printing every n^{th} dot with all zero's in the fire select shift register

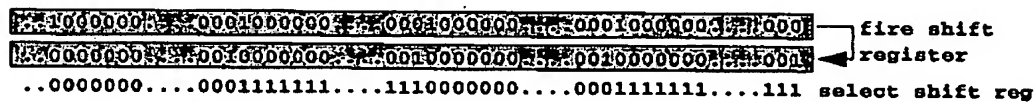b) Printing every n^{th} dot with all one's in the fire select shift register

c) Printing every n^{th} dot with n zero's then n one's in the fire select shift registers

This is done by firing 2 nozzles in every $2n$ group of nozzle at the same time starting outer 2 nozzles working towards the centre two (or the starting from the centre, and working towards the outer two) at the fire rate controlled by *FrClk*.
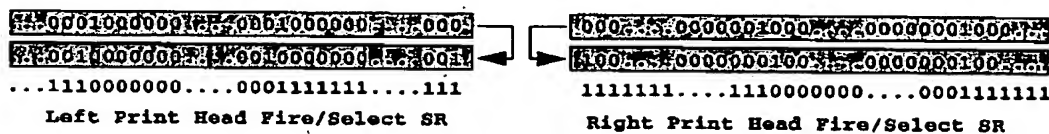
To achieve this fire pattern the **fire shift register** and **select shift register** need to be set up as show in Figure 6.

FIGURE 6. Fire and Select Shift Register setup for printing


```
..1000000....0001000000....0000100000....0001000000....000  ┐ fire shift
..0000000....001000000....0001000000....0010000000....000  ◄─┘ register
..0000000....0001111111....1110000000....0001111111....111  select shift reg
```

The pattern has shifted a '1' into the **fire shift register** every n^{th} positions (where n is usually is a minimum of about 100) and n '1's, followed n '0's in the **select shift register**. At a start of a print cycle, these patterns need to be aligned as above, with the "1000..." of a forward half of **fire shift register**, matching an n grouping of '1' or '0's in the **select shift register**. As well, with the "1000..." of a reverse half of the **fire shift register**, matching an n grouping of '1' or '0's in the **select shift register**. And to continue this print pattern across the butt ends of the chips, the **select shift register** in each should end with a complete block of n '1's (or '0's).

FIGURE 7. Fire Pattern across butt end of Print Chips


```
...1110000000....0001111111....111          1111111....1110000000....0001111111
   Left Print Head Fire/Select SR              Right Print Head Fire/Select SR
```

Since the two chips can be of different lengths, it makes initialisation of these pattern difficult. This is solved by building initialisation circuitry into chips. This circuit is controlled by to registers, **nlen(14)** and **count(14)** and **b(1)**. These registers are loaded serially through *Data[0]*, while *LSyncL* is low, and *ReadL* is high with *FrClk*.

FIGURE 8. Fire Pattern Generation
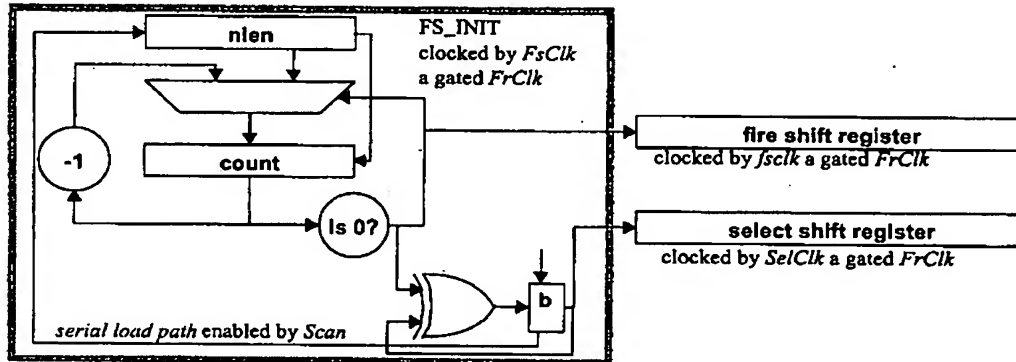
The scan order from input is **b, n[13-0],c[0-13]**, therefore **b** is shifted in last.

The following table shows the values to programme the bi-lithic head pairs using a fire

TABLE 4. Head Combinations Initialisation for $n=100$

| Nozzles L_A	Nozzles L_B	$nlen_{(A\&B)} = n-1$	$count_A = (L_A/2) \bmod n -1$	b_A	b_B	$rem = (L_B/2) \bmod n$	$count_B = (L_A-L_B+rem) \bmod n -1$
9744	4080	99	71	0	0	40	3
8328	5496	99	63	0	0	48	79
6912	6912	99	55	0	0	56	55

pattern length of 100. The calculation assumes head 'A' is the longest head of the pair and once the registers are initialised with LA FrClk cycles (ReadL='0', LSyncL='1'). *rem* would be the correct value for **count$_B$** if chip B was only clocked (FrClk) L_B times. But this chip will be over clocked L_A-L_B cycles. The values of b_A and b_B are either the same or inverse of each other. The actually value does not matter. They need to be different from each other if the **select shift registers** would end up with different values at the butt ends. If $(L_A/2n)$ is even (and **count$_A$** is non zero), then the final run in 'A's select shift register will be !b_A. If $(L_A-L_B/2) \bmod n$ is even (and **count$_B$** is non zero) then the final run in 'B's select shift register will be !b_B.

FIGURE 9. Determining Select Shift Register value
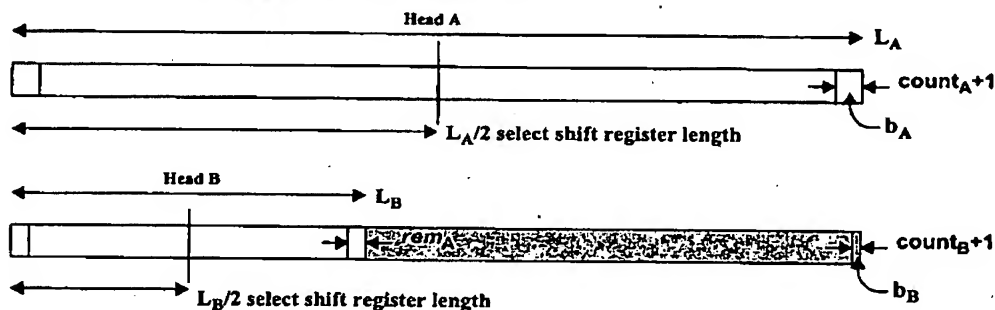
3.4 Profile Pattern

A profile pattern is repeated at *FrClk* rate. It is expected to be a single pulse about 1us long. But it could be a more complicated series of pulse. The actual pattern depends on the ink type.

The following figure show the external timing to print a line of data. In this example the line is printed in 8 cycles of *FrClk*.

FIGURE 10. Timing for printing Signals
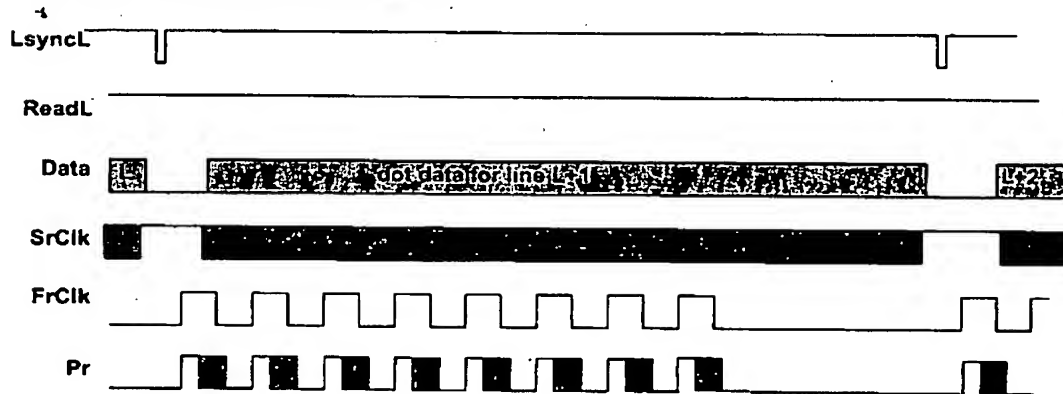
3.5 Interface Modes

The print heads a eight different modes controlled by signals *ReadL* and *LSyncL*. As seen in Figure 9 with both *LSyncL* and *ReadL* high, the chip in normal printing mode. Some of these mode can operate at the same time, but may interfere with the result of the other modes.

TABLE 5. Print Head Modes

ReadL	LSyncL	Mode	Internal Mapping
1	1	Normal Print Mode	SrClk=SrClk/3 frclk=FrClk SelClk=0 FsClk=FrClk Scan=0 CoreScan=0
X	0	Dot Load Mode • **Dot latches** are open, loaded with **Dot shift regis-.ters**, latch once *LSyncL* returns to 1 (this happens regardless of *ReadL*) • Enables Dot Shift register to capture fire result.	
1	0	Fire Load Mode • *Data[0]* will shift through **nlen**, **count** and **b** with ·*FrClk*	SrClk=X frclk=X SelClk=X FsClk=FrClk Scan=1 CoreScan=X

TABLE 5. Print Head Modes

ReadL	LSyncL	Mode	Internal Mapping
0	1	Reset Nozzle Test • Resets the state of nozzle test circuit	SrClk=SrClk FrClk=FrClk SelClk=FrClk
0	1	CMOS testing mode • The contents of the dot shift registers are serial shifted out on *Data[0-1]* with *SrClk*	FsClk=FrClk Scan=0 CoreScan=1
0	1	Fire Initialise mode • The contents of the **fire shift register** and **select shift register** is generated with *FrClk*	
0	0	Temperature Output • The series of Delta Sigma output are clocked out on *Data[0]* with *FrClk*. The sum of these bits represent the temperature of the chip.	SrClk=X frclk=0 SelClk=0 FsClk=0 Scan=0
0	0	Nozzle Test Output • The result of a nozzle test is output on Data[1].	CoreScan=X

3.5.1 Printing

Figure 10 shows show timing for normal printing. During this action, we drop out of *Normal Print Mode*, to *Dot Load Mode* between line transfers. For printing to perform correctly, no other signal should be stable.

3.5.2 Initialising for Printing

To initialise for printing the fire shift registers and select shift registers need to setup into a state as shown in Figure 7. To do this the chips are put into *Fire Load Mode* and the values for **nlen**, **count** and **b** are serially shifted from Data[0] clocked by *FrClk*. As the two chip have separate Data line, and common FrClk, this happens at the same time. Once this is done, mode is changed to *Fire Initialise Mode*, and further L_A *FrClk* cycles are provided to both chips. During all these operation *Pr* should be low, to prevent unintentional firing for nozzles.

FIGURE 11. Initialising Print Heads

Fire Load Mode   *Fire Initialise Mode*
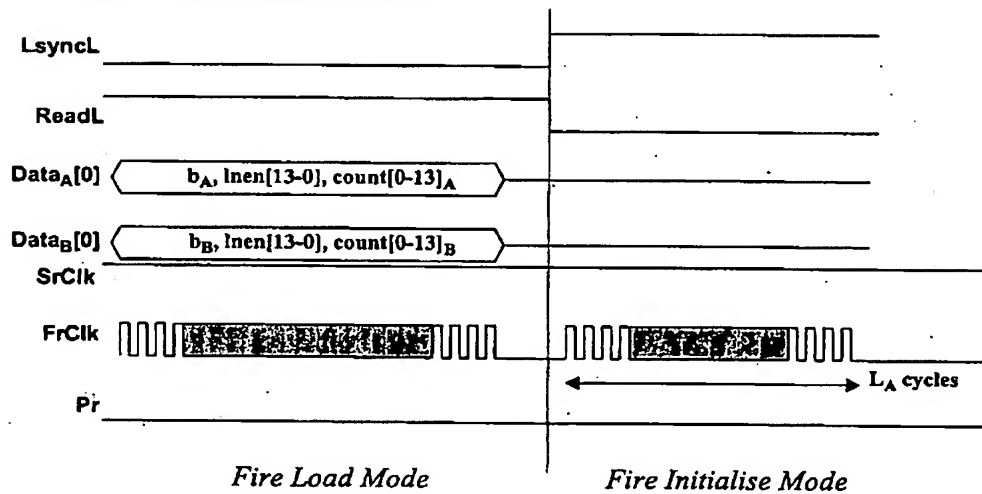
3.5.3 Nozzle Testing

Nozzle testing is done by firing a single at a time a monitoring the *Data[1]* pin in the *Nozzle Test Output* mode.

Each nozzle has a test switch with closes when it nozzle is fired. All 12 switches in a nozzle column are connect in parallel to the following circuit.

FIGURE 12. Nozzle Test Latching Circuit
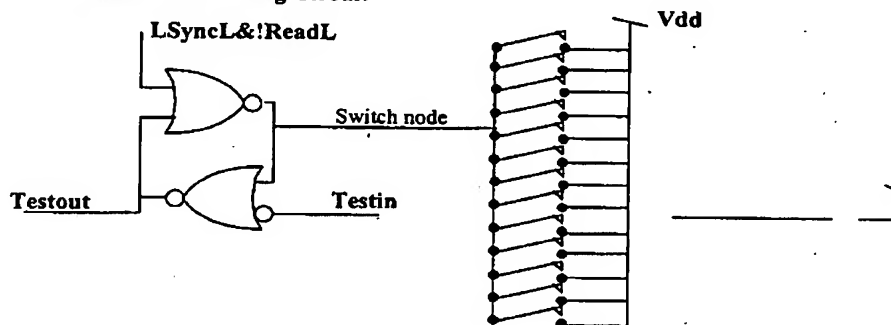
This circuit is initialised when ever LSyncL is high and ReadL is low (*Reset Nozzle Test* mode). This forces all "switch nodes" to low, and the feedback through lower NOR gate will latches this value. With *LSyncL* low and *ReadL* still low (*Nozzle Test Output* mode) the Testout of the first nozzle column is output on *Data[1]*. If any switch is closed, the switch node of this column will be pulled up, and will ripple through to the output as transition from high to low.

FIGURE 13. Nozzle Testing

| Set up Test | Reset Nozzle Test Mode | Nozzle Test Output Mode | Set up Test |
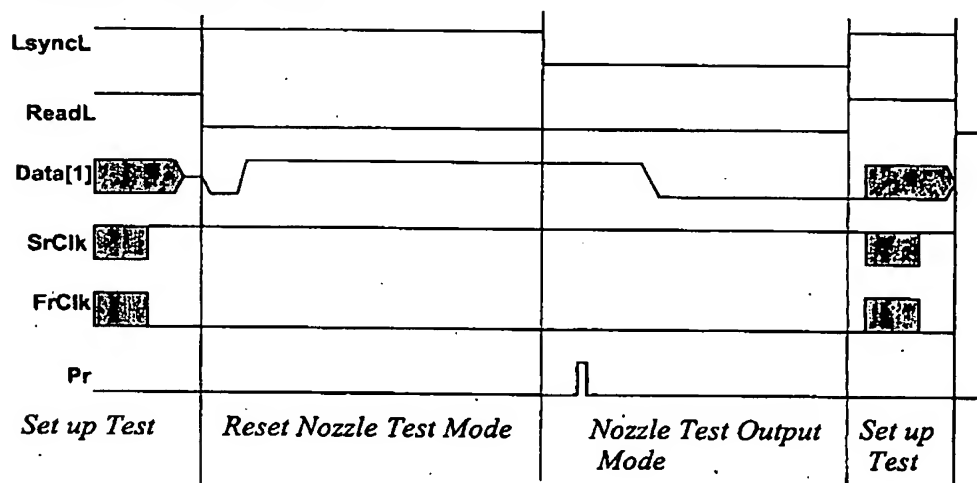
Nozzle testing requires a setup phase in order to fire only one nozzle. There are many ways to achieve this. Simplest might be to load a single colour with 101010 through the even nozzles, and 010101... for the odd nozzles (0's for all other colours), and set up a fire pattern with $n = L_A/2$. With this fire pattern only one nozzle will fire in each Pr pulse. After firing in *Nozzle Test Output* mode, a single *FrClk* will advance to next nozzle, then *Reset* and *Test*. After $L_A/2$ cycles of this testing, a single *SrClk* will advance the dot shift registers to setup the untested nozzles of this colour, and another $L_A/2$ cycles of *FrClk*, *Reset* and *Test* will finished testing this colour. Then repeat test procedure for other colours.

3.5.4 Temperature Output

This mode is not well defined yet. In this mode, *Data[0]* will output a series of ones and zeros clocked by *FrClk*. After a (currently unknown) number of *FrClk* cycles the sum of this series will represent the temperature of the chip. Clocking frequency in this mode it expected to be in the range 10kHz - 1MHz.

FIGURE 14. Temperature Reading
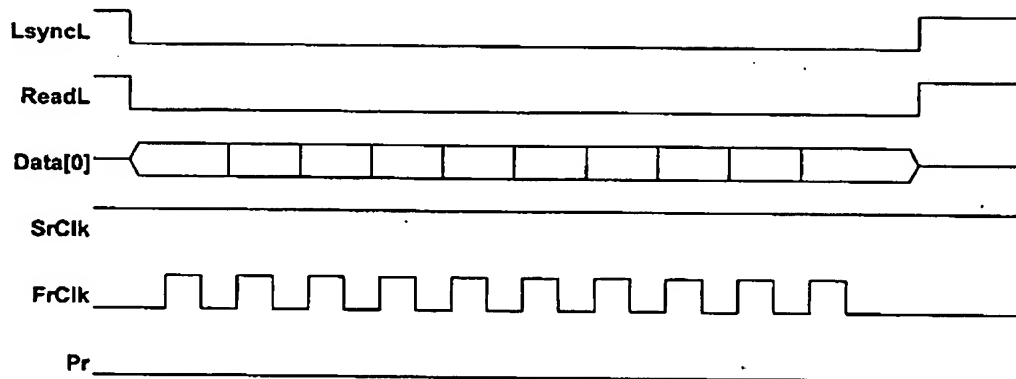
The Frequency of *FrClk* and the number of cycles need to be programmable. Since this mode cycles *FrClk*, the result of fire shift register and select shift register would be changed, but in this mode *FrClk* is disabled to these circuit. So printing can resume without reinitialising.

3.5.5 CMOS Testing

CMOS testing is a mode meant for chip testing with before MEMS as added to the chip. This mode allows the dot shift register to be shifted out on the *Data[0-1]* pins. Much like the *nozzle test mode*, the nozzles are fired while *LSyncL* is low, but during the firing *SrClk* will be cycle, and the **dot shift register** will load the signal that would fire the nozzle. Once capture, the result can be shifted out.

FIGURE 15. CMOS Testing
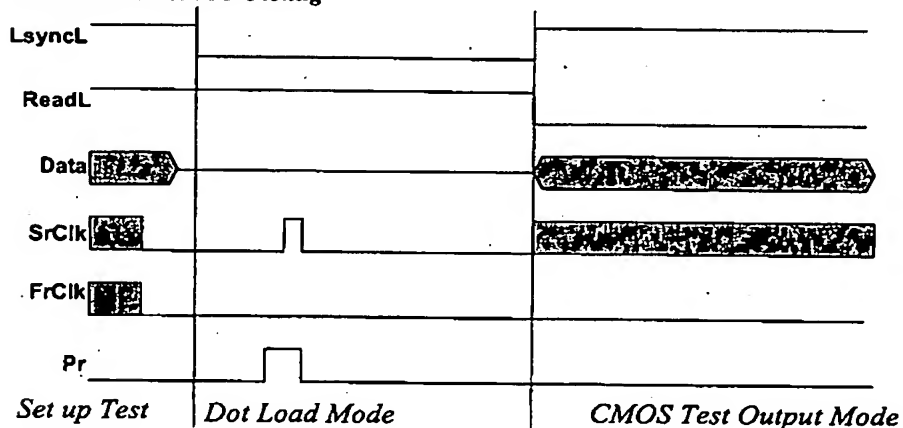
The *Dot Load Mode* above violates normal printing procedure by firing the nozzles (*Pr*) and modify the **dot shift register** (*SrClk*).

4.0 Reticle Layout

To make long chips we need to stitch the CMOS (and MEMS) together by overlapping the reticle stepping field. The reticle will contain two areas:

FIGURE 16. Reticle Layout
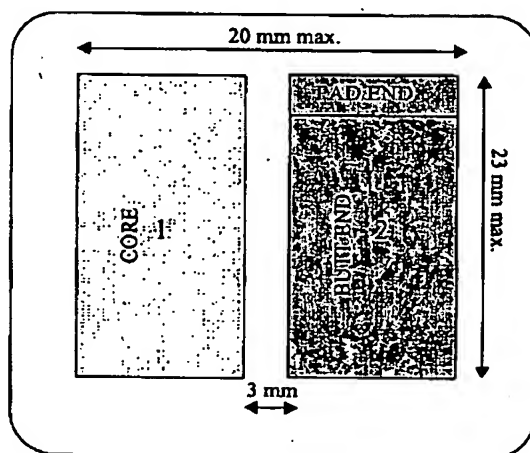
The top edge of *Area 2*, PAD END contains the pads that stitch on bottom edge of *Area 1*, CORE. *Area 1* contains the core array of nozzle logic. The top edge of *Area 1* will stitch to the bottom edge of itself. Finally the bottom edge of *Area 2*, BUTT END will stitch to the top edge of *Area 1*. The BUTT END to used to complete a feedback wiring and seal the chip.

The above region will then be exposed across a wafer bottom to top. *Area 2, Area 1, Area 1...., Area 2.* Only the PAD END of *Area 2* needs to fit on the wafer. The final exposure fo Area 2 only requires the BUTT END on the wafer.

FIGURE 17. Stepper Pattern on Wafer
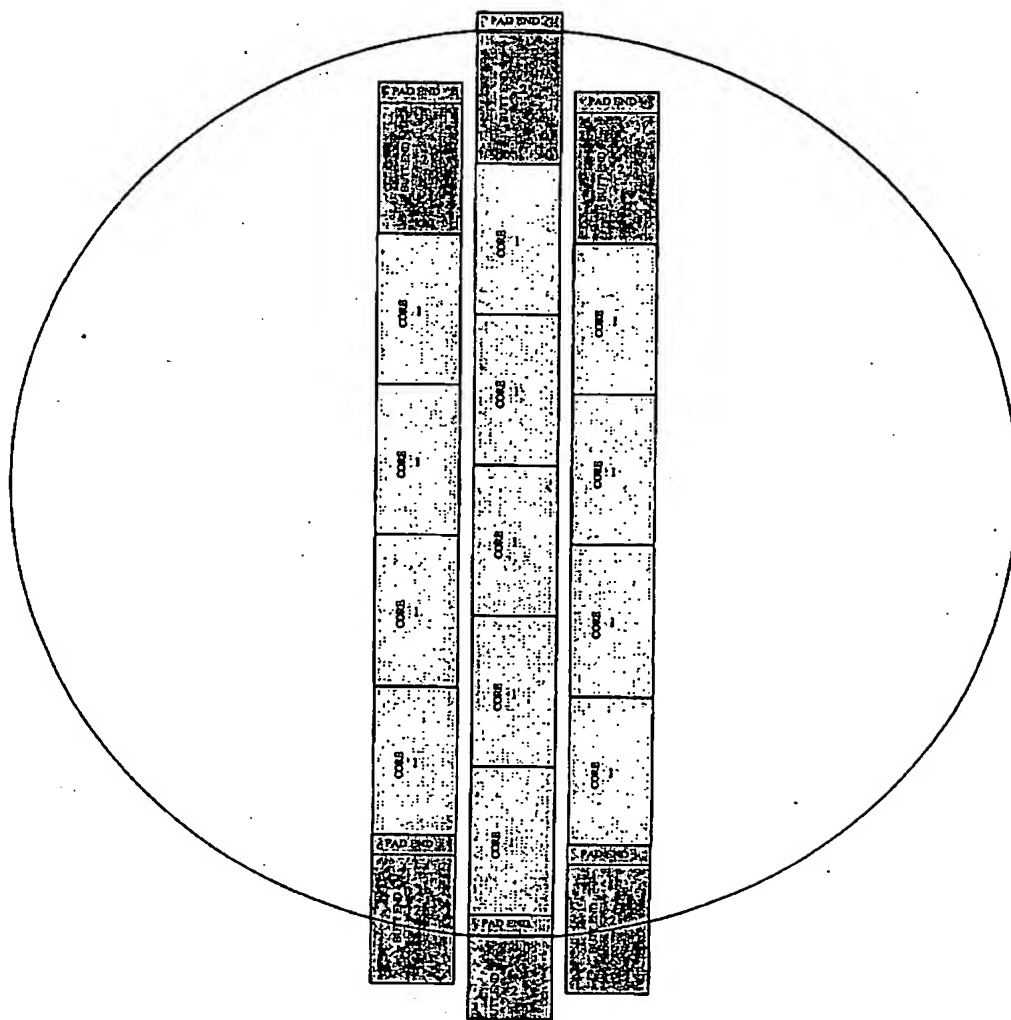
4.1 TSMC U-Frame requirements.

TSMC will be building us frames 10 mm x 0.23 mm which will be placed either side of both *Area 1* and *Area 2*.

TSMC requires 6 mm area for blading between the two exposure area. This translates to 3 mm on the reticle, as some recticles are 2x size, while most are 5x, the worst case must be used.

SoPEC

Security Overview

4-4-1-3 version 1.6

November 29, 2002

Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: Info@SilverbrookResearch.com

1   Introduction

1.1   DOCUMENT HISTORY

Version	Date	Authors	Details
1.6	29 November, 2002	Simon Walmsley	Updated ChipA to be ChipR to match protocols document, got rid of 68k reference now that we are using LEON.
1.5	26 November, 2002	Simon Walmsley	Added description of storing more than a single SoPEC_id key in a PRINTER_QA (in section 3.5.3 and related). This reduces the cost of a multi-SoPEC system with no loss of security. Also added text to describe that batch keys can be different for each SoPEC if the indirect upgrade key protocol is used.
1.4	9 September, 2002	Simon Walmsley	Added section in requirements detailing types of attacks we care about and don't care about.
1.3	30 August, 2002	Simon Walmsley	Changed ComCo_OEM_xxxx variables into simply xxxx variables, since that is more generic. Added text regarding ink refill. Added extra software authentication stage to prevent ComCos from fiddling with SoPEC software.
1.2	29 August, 2002	Simon Walmsley	Added section on how the PRINTER_QA chip gets programmed with the SoPEC_id_key.
1.1	28 August, 2002	Simon Walmsley	Updated to have ink and operating parameters be authenticated via symmetric key based signatures based on a unique SoPEC_id.
1.0	27 August, 2002	Simon Walmsley	Updated after review.
0.2 draft	26 August, 2002	Simon Walmsley	Changed public-key and private key references to asymmetric & symmetric respectively, so private can now sub-refer to the private key of the asymmetric pair, or the single private symmetric key. Changed OEM_id into ComCo_OEM_license_id to more accurately reflect the scope of the id.
0.1 draft	26 August, 2002	Simon Walmsley	Initial issue.

1.2   REFERENCES

[1]   Silicon & Software Systems, *4-4-9-4 SoPEC Hardware Design*.
[2]   Silverbrook Research, *4-2-1-1 Print Engine Controller Hardware Design*.
[3]   Silverbrook Research, *4-3-1-2 QA Chip Technical Reference*.
[4]   Silverbrook Research, *4-3-1-8 QA Chip Programmer Requirements*.
[5]   Silverbrook Research, *4-3-1-26 Authentication Protocols*.

1.3   SCOPE

This document describes the basic security requirements of programs running on the SoPEC ASIC [1]. It then describes an implementation solution to the security requirements.

The described solution impacts the design of the SoPEC ASIC as well as implying key management issues. The solution includes references to the QA Chip ASIC [3] and associated authentication protocols [5].

It is possible that some of the requirements and defined solution will be applicable to systems built with the PEC ASIC [2], although such systems are beyond the scope of this document.

1.4   READERSHIP

This document is written for software engineers and system architects that are working with SoPEC, as well as PCB designers that are responsible for SoPEC-based Print Engines. A similar audience working on PEC and PEC-based Print Engines may also find document useful.

This document is also intended to be read by those responsible for key management and associated database designers with regards to guiding requirements.

This document is confidential to Silverbrook Research Pty. Ltd. and its distribution outside this organisation *must* be covered by a non-disclosure agreement (NDA).

1.5   QA CHIP TERMINOLOGY

The Authentication Protocols document [5] refers to QA Chips by their function in particular protocols:

- For authenticated reads, ChipR is the QA Chip being read from, and ChipT is the QA Chip that identifies whether the data read from ChipR can be trusted.
- For replacement of keys, ChipP is the QA Chip being programmed with the new key, and ChipF is the factory QA Chip that generates the message to program the new key.
- For upgrades of data in memory vectors, ChipU is the QA Chip being upgraded, and ChipS is the QA Chip that signs the upgrade value.

Any given physical QA Chip will contain functionality that allows it to operate as an entity in some number of these protocols.

Therefore, wherever the terms ChipR, ChipT, ChipP, ChipF, ChipU and ChipS are used in this document, they are referring to *logical* entities involved in an authentication protocol as defined in [5].

Physical QA Chips are referred to by their location. For example, each ink cartridge may contain a QA Chip referred to as an INK_QA, with all INK_QA chips being on the same physical bus. In the same way, the QA Chip inside the printer is referred to as PRINTER_QA, and will be on a separate bus to the INK_QA chips.

2   Requirements

2.1   SECURITY

The basic functional security requirements are:

- Silverbrook code and OEM program code co-existing safely
- Silverbrook operating parameters authentication
- OEM operating parameters authentication
- Ink usage authentication

Each of these is outlined in subsequent sections.

The authentication requirements imply that:

- OEMs and end-users must not be able to replace or tamper with Silverbrook program code or data
- OEMs and end-users must not be able to call unauthorized functions within Silverbrook code
- End-users must not be able to replace or tamper with OEM program code or data
- End-users must not be able to call unauthorized functions within OEM program code
- OEMs must be able to test products at their highest upgradable status, yet not be able to ship them outside the terms of their license
- OEMs and end-users must not be able to directly access the print engine pipeline (PEP) hardware, the LSS Master (for QA Chip access) or any other peripheral block with the exception of operating system permitted GPIO pins and timers.

2.1.1   Silverbrook code and OEM program code co-existing safely

SoPEC includes a CPU that must run both Silverbrook program code and OEM program code. The execution model envisaged for SoPEC is one where Silverbrook program code forms an operating system (O/S), providing services such as controlling the print engine pipeline, interfaces to communications channels etc. The OEM program code must run in a form of user mode, protected from harming the Silverbrook program code. The OEM program code is permitted to obtain services by calling functions in the O/S, and the O/S may also call OEM code at specific times. For example, the OEM program code may request that the O/S call an OEM interrupt service routine when a particular GPIO pin is activated.

A basic requirement then, for SoPEC, is a form of protection management, whereby Silverbrook and OEM program code can co-exist without the OEM program code damaging operations or services provided by the Silverbrook O/S. Since services rely on SoPEC peripherals (such as SCB, LSS Master, Timers etc) access to these peripherals should also be restricted to Silverbrook program code only.

2.1.2   Silverbrook operating parameters authentication

A particular OEM will be licensed to run a Print Engine with a particular set of operating parameters (such as print speed or quality). The OEM and/or end-user can upgrade the operating license for a fee and thereby obtain an upgraded set of operating parameters.

Neither the OEM nor end-user should be able to upgrade the operating parameters without paying the appropriate fee to upgrade the license. Similarly, neither the OEM nor end-user

should be able to bypass the authentication mechanism via any program code on SoPEC. This implies that OEMs and end-users must not be able to tamper with or replace Silverbrook program code or data, nor be able to call unauthorized functions within Silverbrook program code.

However, the OEM must be capable of assembly-line testing the Print Engine at the upgraded status before selling the Print Engine to the end-user.

2.1.3   OEM operating parameters authentication

The OEM may provide operating parameters to the end-user independent of the Silverbrook operating parameters. For example, the OEM may want to sell a franking machine[1].

The end-user should not be able to upgrade the operating parameters without paying the appropriate fee to the OEM. Similarly, the end-user should not be able to bypass the authentication mechanism via any program code on SoPEC. This implies that end-users must not be able to tamper with or replace OEM program code or data, as well as not be able to tamper with the PEP blocks or service-related peripherals.

2.1.4   Ink usage authentication

Each OEM sells printers and ink to end-users according to a business model. For example, OEM_1 may provide ink at $A for a $B printer, while OEM_2 may sell the same featured printer at a higher price $A+$X, and provide the ink at a cheaper price $B-$Y. OEM_1 has a business model that relies on the fact that end-users of OEM_1 printers can only use OEM_1 ink, and likewise OEM_2 has a business model that relies on the fact that end-users of OEM_2 printers can only use OEM_2 ink.

It is in the interest of both OEM_1 and OEM_2 that end-users cannot subvert the authentication mechanism for ink. Otherwise the business models are compromised.

It is also in the interests of the Memjet Group that OEM_1 and OEM_2 cannot subvert the authentication mechanism for ink, since the Memjet Group provides OEMs with printers under a license agreement that the OEM will purchase ink from a designated ink supplier.

2.2   ACCEPTABLE COMPROMISES

Since there is no protection physically built into the Memjet printheads, it is theoretically possible for someone (with enough time, money and incentive) to remove the printheads from the print engine, build their own SoPEC ASIC equivalent, write their own program code etc. It is impossible to guard against such an attack.

We are really only concerned with commercial attacks, where there is a total compromise of printer operating parameter authentication and ink usage authentication. An example of such an attack is where the Silverbrook printing O/S is replaced by one that can be downloaded from the internet, and this clone O/S allows usage of the print engine outside the license agreement. Whether the clone O/S is developed by a hacker or by a rogue OEM is not important - it matters that any user can trivially upgrade the printer outside the terms of the license agreement.

---

1. a franking machine prints stamps

If an end user takes the time and energy to hack the print engine and thereby succeeds in upgrading the single print engine only, yet not be able to use the same keys etc on another print engine, that is an acceptable security compromise. However it doesn't mean we have to make it totally simple or cheap for the end-user to accomplish this.

Software-only attacks are the most dangerous, since they can be transmitted via the internet and have no perceived cost. Physical modification attacks are far less problematic, since most printer users are not likely to want their print engine to be physically modified. This is even more true if the cost of the physical modification is likely to exceed the price of a legitemate upgrade.

Finally, it should be noted that all OEMs are bound by license agreements that specify penalties if they attempt to reverse engineer or bypass the print engines. In countries where these agreements are enforceable by law, this at least provides a modicum of security.

2.3   IMPLEMENTATION CONSTRAINTS

Any solution to the requirements detailed in Section 2.1 must also meet certain implementation constraints. These are:

- No flash memory inside SoPEC
- SoPEC must be simple to verify
- Silverbrook program code must be updateable
- OEM program code must be updateable
- Must be bootable from activity on USB or ISI
- No extra pins for assigning IDs to slave SoPECs
- Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)
- Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)
- Cannot trust the ISI comms channel

These constraints are detailed below.

2.3.1   No flash memory inside SoPEC

SoPEC is intended to be implemented in 0.13 micron or smaller. Flash memory will not be available in any of the target processes being considered. Although Virage have a process independent flash cell, it is very large and effectively impractical for anything more than a few bits.

2.3.2   SoPEC must be simple to verify

All combinatorial logic and embedded program code within SoPEC must be verified before manufacture. Every increase in complexity in either of these increases verification effort and increases risk.

2.3.3   Silverbrook program code must be updateable

It is not possible nor even desirable to write a single complete operating system that is:

- verified completely (see Section 2.3.1)
- correct for all possible future uses of SoPEC systems
- finished in time for SoPEC manufacture

Therefore the complete Silverbrook program code must not *permanently* reside on SoPEC. It must be possible to update the Silverbrook program code as enhancements to functionality are made and bug fixes are applied.

In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Silverbrook.

2.3.4   OEM program code must be updateable

Given that each OEM will be writing specific program code for printers that have not yet been conceived, it is impossible for all OEM program code to be embedded in SoPEC at the ASIC manufacture stage.

Since flash memory is not available (see Section 2.3.1), OEMs cannot store their program code in on-chip flash. While it is theoretically possible to store OEM program code in ROM on SoPEC, this would entail OEM-specific ASICs which would be prohibitively expensive. Therefore OEM program code cannot *permanently* reside on SoPEC.

Since OEM program code must be downloadable for SoPEC to execute, it should therefore be possible to update the OEM program code as enhancements to functionality are made and bug fixes are applied.

In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Silverbrook.

2.3.5   Must be bootable from activity on USB or ISI

SoPEC can be placed in sleep mode to save power when printing is not required. RAM is not preserved in sleep mode. Therefore any program code and data in RAM will be lost. However, SoPEC must be capable of being woken up from the host when it is time to print again.

In the case of a single SoPEC system, the host communicates with SoPEC via USB.

In the case of a multi-SoPEC system, the host typically communicates with the ISI Master chip (e.g. the ISI Master could be SoPEC, and the comms is USB), and can send messages to other slave SoPECs via the ISI master. The ISI master SoPEC relays these messages to the slaves via the ISI.

Therefore SoPEC must be capable of being woken up by activity on either the USB or on the ISI.

2.3.6   No extra pins to assign IDs to slave SoPECs

In a single SoPEC system the host only sends data to the single SoPEC. However in a multi-SoPEC system, each of the slaves needs to be uniquely identifiable in order to be able for the host to send data to the correct slave.

Since there is no flash on board SoPEC (Section 2.3.1) we are unable to store a slave ID (eg 4 bits) in each SoPEC. Moreover, any ROM in each SoPEC will be identical.

It is possible to assign n pins to allow 2^n combinations of IDs for slave SoPECs. However a design goal of SoPEC is to minimize pins for cost reasons, and this is particularly true of features only used in multi-SoPEC systems. We have 2 pins for inter-SoPEC communications, and further pins would add to the cost.

The design constraint requirement is therefore to allow slaves to be IDed via a method that does not require any extra pins. This implies that whatever boot mechanism that satisfies the security requirements of Section 2.1 must also be able to assign IDs to slave SoPECs.

2.3.7   Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)

If the printer operating parameters are stored in the non-volatile memory of the Print Engine's on-board PRINTER_QA chip, both Silverbrook and OEM program code cannot rely on the communication channel being secure. It is possible for an end-user to replace the PRINTER_QA chip or subvert the communications channel.

2.3.8   Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)

The amount of ink remaining for a given ink cartridge is stored in the non-volatile memory of that ink cartridge's INK_QA chip. Both Silverbrook and OEM program code cannot rely on the communication channel to the INK_QA being secure. It is possible for an end-user to replace the INK_QA chip or subvert the communications channel.

2.3.9   Cannot trust the ISI comms channel

In a multi-SoPEC system, or in a single-SoPEC system that has a non-USB connection to the host, a given SoPEC will receive its data over the ISI. It is quite possible for an end-user to insert a chip that subverts the communications channel (for example performs man-in-the-middle attacks).

3   Proposed Solution

A proposed solution to the requirements of Section 2, can be summarised as:

- Each SoPEC has a unique id
- CPU with user/supervisor mode
- Memory Management Unit
- Specific entry points in O/S
- Boot procedure, including authentication of program code and operating parameters
- SoPEC ISI identification

3.1   EACH SOPEC HAS A UNIQUE ID

Each SoPEC needs to contains a unique *SoPEC_id* of minimum size 64-bits[1]. This *SoPEC_id* is used to form a symmetric key unique to each SoPEC: *SoPEC_id_key*.

The verification of operating parameters and ink usage depends on *SoPEC_id* being diffi-cult to determine. Difficult to determine means that someone should not be able to deter-mine the id via software, or by viewing the communications between chips on the board. If the *SoPEC_id* is available through running a test procedure on specific test pins on the chip, then depending on the ease by which this can be done, it is likely to be acceptable.

It is important to note that in the proposed solution, compromise of the *SoPEC_id* leads only to compromise of the operating parameters and ink usage on this particular SoPEC. It does not compromise any other SoPEC or all inks or operating parameters in general.

It is ideal that the *SoPEC_id* be random, although this is unlikely to occur on standard manufacture processes for ASICs. If the id is within a small range however, it will be able to be broken by brute force. This is why 32-bits is not sufficient protection.

3.2   CPU WITH USER/SUPERVISOR MODE

SoPEC contains a CPU with direct hardware support for user and supervisor modes. At present, the intended CPU is the LEON (a 32-bit processor with an instruction set accord-ing to the IEEE-1754 standard. The IEEE1754 standard is compatible with the SPARC V8 instruction set).

Silverbrook (operating system) program code will run in supervisor mode, and all OEM program code will run in user mode.

3.3   MEMORY MANAGEMENT UNIT

SoPEC contains a Memory Management Unit (MMU) that limits access to regions of DRAM by defining read, write and execute access permissions for supervisor and user mode. Program code running in user mode is subject to user mode permission settings, and program code running in supervisor mode is subject to supervisor mode settings.

---

1. On IBM's CU11 process this chipId is 80 bits.

A setting of 1 for a permission bit means that type of access (e.g. read, write, execute) *is* permitted. A setting of 0 for a read permission bit means that that type of access is *not* permitted.

At reset and whenever SoPEC wakes up, the settings for all the permission bits are 1 for all supervisor mode accesses, and 0 for all user mode accesses. This means that supervisor mode program code must explicitly set user mode access to be permitted on a section of DRAM.

Access permission to all the non-valid address space should be trapped, regardless of user or supervisor mode, and regardless of the access being read, execute, or write.

Access permission to all of the valid non-DRAM address space (for example the PEP blocks) is supervisor read / write access only (no supervisor execute access, and user mode has no acccess at all) with the exception that certain GPIO and Timer registers can also be accessed by user code. These registers will require bitwise access permissions. Each peripheral block will determine how the access is restricted.

The embedded DRAM should start at 0x0000_0000 to support programmable exception vectors. The reset exception vector (and possibly some others) is translated in the MMU to point to the appropriate location in ROM, ideally in a manner that still allows null pointer dereferencing to be trapped.

With respect to the DRAM and PEP subsystems of SoPEC, typically we would set user read/write/execute mode permissions to be 1/1/0 only in the region of memory that is used for OEM program data, 1/0/1 for regions of OEM program code, and 0/0/0 elsewhere. By contrast we would typically set supervisor mode read/write/execute permissions for this memory to be 1/1/0 (to avoid accidentally executing user code in supervisor mode).

The *SoPEC_id* parameter (see Section 3.1) should only be accessible in supervisor mode, and should only be stored and manipulated in a region of memory that has no user mode access.

3.4   SPECIFIC ENTRY POINTS IN O/S

Given that user mode program code cannot even call functions in supervisor code space, the question arises as how OEM programs can access functions, or request services. The implementation for this depends on the CPU.

On the LEON processor, the TRAP instruction allows programs to switch between user and supervisor mode in a controlled way. The TRAP switches between user and supervisor register sets, and calls a specific entry point in the supervisor code space in supervisor mode. The TRAP handler dispatches the service request, and then returns to the caller in user mode.

Use of a command dispatcher allows the O/S to provide services that filter access - e.g. a generalised print function will set PEP registers appropriately and ensure QA Chip ink updates occur.

The LEON also allows supervisor mode code to call user mode code. There are a number of ways that this functionality can be implemented.

3.5   BOOT PROCEDURE

3.5.1   Basic premise

The intention is to load the Silverbrook and OEM program code down into SoPEC's RAM, where it can be subsequently executed. The basic SoPEC therefore, must be capable of downloading program code. However SoPEC must be able to guarantee that only authorized Silverbrook boot programs can be downloaded, otherwise anyone could modify the O/S to do anything, and then download that - thereby bypassing the licensed operating parameters.

We perform authentication of program code and data using asymmetric cryptography and *without* using a QA Chip.

Assuming we have already downloaded some data and a 160-bit signature into eDRAM, the boot loader needs to perform the following tasks:

- perform SHA-1 on the downloaded data to calculate a digest *localDigest*
- perform asymmetric decryption on the downloaded signature (160-bits) using an asymmetric public key to obtain *authorizedDigest*
- If *localDigest* = *authorizedDigest*, then the downloaded data is authorized (the signature must have been signed with the asymmetric private key) and control can then be passed to the downloaded data

Asymmetric decryption is used instead of symmetric decryption because the decrypting key must be held in SoPEC's ROM. If symmetric private keys are used, the ROM can be probed and the security is compromised.

The procedure requires the following data item:

- **boot0key** = an *n*-bit asymmetric public key

The procedure also requires the following two functions:

- **SHA-1** = a function that performs SHA-1 on a range of memory and returns a 160-bit digest
- **decrypt** = a function that performs asymmetric decryption of a message using the passed-in key

Assuming that all of these are available (e.g. in the boot ROM), boot loader 0 can be defined as in the following pseudocode:

```
bootloader0(data, sig)
   localDigest ← SHA-1(data)
   authorizedDigest ← decrypt(sig, boot0key)
   if (localDigest = authorizedDigest)
      jump to program code at data-start address// will never to return
   Else
      // program code is unauthorized
   EndIf
```

The length of the key will depend on the asymmetric algorithm chosen. The key must provide the equivalent protection of the entire QA Chip system - if the Silverbrook O/S program code can be bypassed, then it is equivalent to the QA Chip keys being compromised. In fact it is worse because it would compromise Silverbrook operating parameters, OEM operating parameters, and ink authentication by software downloaded off the net (e.g. from some hacker in Norway).

In the case of RSA, a 2048-bit key is required to match the 160-bit symmetric-key security of the QA Chip. In the case of ECDSA, a key length of 132 bits is likely to suffice.

There is also no advantage to storing multiple keys in SoPEC and having the *external* message choose which key to validate against, because a compromise of any key allows the external user to always select that key. Even if there were 2 keys, one for USB-based booting and one for ISI-based booting, a compromise of the USB-based booting key is enough to compromise all the single SoPEC systems (99% of SoPEC systems).

Therefore the entire security of SoPEC is based on keeping the asymmetric private key paired to boot0key secure. The entire security of SoPEC is also based on keeping the program that signs (i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.

If a compromise is discovered, it may be economically feasible to change the *boot0key* value in SoPEC's ROM, since this is only a single mask change, and would be easy to verify and characterize.

3.5.2   Hierarchies of authentication

Given that test programs, evaluation programs, and Silverbrook O/S code needs to be written and tested, and OEM program code etc. also needs to be tested, it is not secure to have a single authentication of a monolithic dataset combining Silverbrook O/S, non-O/S, and OEM program code - we certainly don't want OEM's signing Silverbrook program code, and Silverbrook shouldn't have to be involved with the signing of OEM program code.

Therefore we require differing levels of authentication and therefore a number of keys, although the procedure for authentication is identical to the first - a section of program code contains the key for authenticating the next.

This method allows for any hierarchy of authentication, based on a root key of boot0key.

For example, assume that we have the following entities:

- SoPECCo, Silverbrook's SoPEC hardware / software company. Supplies SoPEC ASICs and SoPEC O/S printing software to a ComCo.
- ComCo, a company that assembles Print Engines from SoPECs, Memjet printheads etc, customizing the Print Engine for a given OEM according to a license
- OEM, a company that uses a Print Engine to create a printer product to sell to the end-users. The OEM would supply the motor control logic, user interface, and casing.

The levels of authentication hierarchy are as follows:

- SoPECCo generates *dataset1*, consisting of the print engine O/S (which incorporates the print engine functionality) and the ComCo's asymmetric public key. SoPECCo signs *dataset0* with SoPECCo's asymmetric private *boot0key* key. The print engine program code expects to see an operating parameter block signed by the ComCo's asymmetric private key.
- The ComCo generates *dataSet3*, consisting of *dataset1* plus *dataset2*, where *dataset2* is an operating parameter block for a given OEM's print engine licence (according to the print engine license arrangement) signed with the ComCo's asymmetric private key. The operating parameter block (*dataset2*) would contain valid print speed ranges, a *PrintEngineLicenseId*, and the OEM's asymmetric public key. The ComCo can gen-

erate as many of these operating parameter blocks for any number of Print Engine Licenses, but cannot write or sign any supervisor O/S program code.

- The OEM would generate *dataset5*, consisting of *dataset3* plus *dataset4*, where *dataset4* is the OEM program code signed with the OEM's asymmetric private key. The OEM can produce as many versions of *dataset5* as it likes (e.g. for testing purposes or for updates to drivers etc)

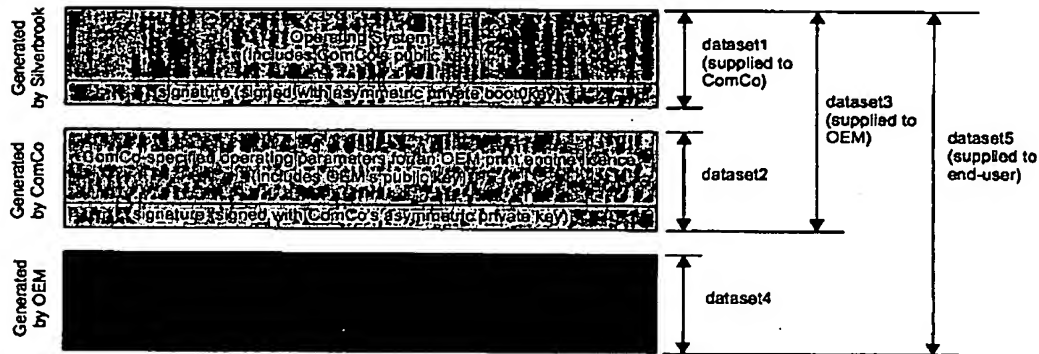The relationship is shown below in Figure 1.

Figure 1. Relationship between the datasets

When the end-user uses *dataset5*, SoPEC itself validates *dataset1* via the *boot0key* mechanism described in Section 3.5.1. Once *dataset1* is executing, it validates *dataset2*, and uses *dataset2* data to validate *dataset4*. The validation hierarchy is shown in Figure 2.
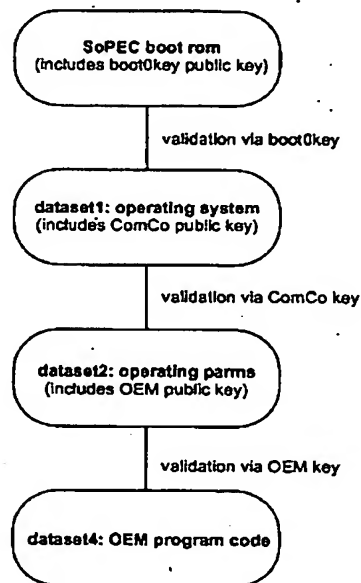

Figure 2. Validation hierarchy

If a key is compromised, it compromises all subsequent authorizations down the hierarchy. In the example from above (and as illustrated in Figure 2) if the OEM's asymmetric private key is compromised, then O/S program code is not compromised since it is above OEM program code in the authentication hierarchy. However if the ComCo's asymmetric private key is compromised, then the OEM program code is also compromised. A compromise of *boot0key* compromises <u>everything</u> up to SoPEC itself, and would require a mask ROM change in SoPEC to fix.

It is worthwhile repeating that in any hierarchy the security of the entire hierarchy is based on keeping the asymmetric private key paired to boot0key secure. It is also a requirement that the program that signs (i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.

3.5.3   Authenticating operating parameters

Operating parameters need to be considered in terms of Silverbrook operating parameters and OEM operating parameters. Both sets of operating parameters are stored on the PRINTER_QA chip (physically located inside the printer). This allows the printer to maintain parameters regardless of being moved to different computers, or a loss/replacement of host O/S drivers etc.

On PRINTER_QA, memory vector M_0 contains the upgradable operating parameters, and memory vectors M_{1+} contains any constant (non-upgradable) operating parameters.

Considering only Silverbrook operating parameters for the moment, there are actually two problems:

a. setting and storing the Silverbrook operating parameters, which should be authorized only by Silverbrook

b. reading the parameters into SoPEC, which is an issue of SoPEC authenticating the data on the PRINTER_QA chip since we don't trust PRINTER_QA.

The PRINTER_QA chip therefore contains the following symmetric keys:

- $K_0 = PrintEngineLicense\_key$. This key is constant for all SoPECs supplied for a given print engine license agreement between an OEM and a Silverbrook ComCo. K_0 has write permissions to the Silverbrook upgradeable region of M_0 on PRINTER_QA.
- $K_1 = SoPEC\_id\_key$. This key is unique for each SoPEC (see Section 3.1), and is known only to the SoPEC and PRINTER_QA. K_1 does not have write permissions for anything.

K_0 is used to solve problem (a). It is only used to authenticate the actual upgrades of the operating parameters. Upgrades are performed using the standard upgrade protocol described in [5], with PRINTER_QA acting as the ChipU, and the external upgrader acting as the ChipS.

K_1 is used by SoPEC to solve problem (b). It is used to authenticate reads of data (i.e. the operating parameters) from PRINTER_QA. The procedure follows the standard authenticated read protocol described in [5], with PRINTER_QA acting as ChipR, and the embedded supervisor software on SoPEC acting as ChipT. The authenticated read protocol [5] requires the use of a 160-bit nonce, which is a pseudo-random number. This creates the problem of introducing pseudo-randomness into SoPEC that is not readily determinable by OEM programs, especially given that SoPEC boots into a known state. One possibility is to use the same random number generator as in the QA Chip (a 160-bit maximal-lengthed linear feedback shift register) with the seed taken from the value in the *WatchDogTimer* register in SoPEC's timer unit when the first page arrives.

Note that the procedure for verifying reads of data from PRINTER_QA does not rely on Silverbrook's key K_0. This means that precisely the same mechanism can be used to read and authenticate the OEM data also stored in PRINTER_QA. Of course this must be done by Silverbrook supervisor code so that *SoPEC_id_key* is not revealed.

If the OEM also requires upgradable parameters, we can add an extra key to PRINTER_QA, where that key is an OEM_key and has write permissions to the OEM part of M_0.

In this way, K_1 never needs to be known by anyone except the SoPEC and PRINTER_QA.

Each printing SoPEC in a multi-SoPEC system need access to a PRINTER_QA chip that contains the appropriate *SoPEC_id_key* to validate ink useage and operating parameters. This can be accomplished by a separate PRINTER_QA for each SoPEC, or by adding extra keys (multiple *SoPEC_id_keys*) to a single PRINTER_QA.

However, if ink usage is not being validated (e.g. if print speed were the only Silverbrook upgradable parameter) then not all SoPECs require access to a PRINTER_QA chip that contains the appropriate *SoPEC_id_key*. Assuming that OEM program code controls the physical motor speed (different motors per OEM), then the PHI within the first (or only) front-page SoPEC can be programmed to accept (or generate) line sync pulses at a particular rate. If line syncs arrived faster than the particular rate, the PHI would generate a buffer underrun. This would mean that even if the motor speed was hacked to be fast, the print will terminate.

3.5.3.1   OEM assembly-line test

As described in Section 2.1.2, Silverbrook operating parameters include such items as print speed, print quality etc. and are tied to a license provided to an OEM. These parameters are under Silverbrook-control. The licensed Silverbrook operating parameters are stored in the PRINTER_QA as described in Section 3.5.3.

However, although an OEM should only be able sell the licensed operating parameters for a given Print Engine, they must be able to assembly-line test[1] the Print Engine with a different set of operating parameters i.e. a maximally upgraded Print Engine.

Several different mechanisms can be employed to allow OEMs to test the upgraded capabilities of the Print Engine. At present it is unclear exactly what kind of assembly-line tests would be performed.

At first thought, it might be considered that a dongle-style approach using a special master PRINTER_QA containing upgraded parameters might be a solution. However, for the SoPEC to accept the parameters as true, the special master PRINTER_QA must contain the appropriate *SoPEC_id_key* (tied to the specific *SoPEC_id* of the SoPEC system under test). Since the OEM can't perform the key upgrade, they must make use of a Silverbrook upgrade mechanism, which implies either a Silverbrook box, or a connection to a Silverbrook machine (e.g. over a net). Neither approaches are good.

---

1. This section is referring to assembly-line testing rather than development testing. An OEM can maximally upgrade a given Print Engine to allow developmental testing of their own OEM program code & mechanics.

If there is no special master PRINTER_QA for testing, then we must make use of special test programs, or storage on the PRINTER_QA, or both. The solution will depend on the test requirements of the OEM.

A simple test program that allows any pages to be printed at full upgrade capability is definitely not secure. If it gets out to the public, it is effectively a free upgrade. Silverbrook would not want the OEM to have such a program.

Likewise, if a test program only printed pages that had been signed with some key, then not only does this change the timing of real pages (since the signature must be verified before printing) but a service must be made available to sign special test images. This may be possible, but it means that Silverbrook must be involved for every time a test image is designed. If Silverbrook gives the OEM a program that generates signatures to avoid this annoyance, then it is the same as giving away the ability to print at full capability.

If the OEM requires tests that are not actually printing dots, then a test harness software loader that looks and behaves like a real Print Engine (including all output signals etc.) at full upgrade capability, except that it does not produce physical dots (i.e. at the very end of the pipeline, the data is masked before being sent out to the printhead). This will produce a timing accurate result, and is the simplest, most effective solution. If the special driver gets out into the public, the user can only print blank pages.

If the OEM requires tests that actually prints dots, there are several possibilities:

  a. A version of the O/S (signed for the OEM) that will only print special Silverbrook test patterns. This may be quite adequate, but it has the disadvantage that OEM test patterns cannot be printed.

  b. A version of the O/S that prints garbage in special places over the test image. Again the has the disadvantage that special OEM test patterns cannot be printed.

  c. A version of the O/S that reads and decrements a DecrementOnly value in PRINTER_QA. If the value before successful decrementing is non-zero, then the O/S will run at full upgrade capability until either a power-loss or a pre-determined number of pages (e.g. agreed to by the OEM and Silverbrook) have been printed. The number to be stored in the PRINTER_QA at initial PRINTER_QA customization may only need to be 1 or 2.

Of these solutions, option (c) is probably the least restrictive to the OEM while still being useful. If the test program gets out, then if the value in PRINTER_QA is 0 after testing, then there is no impact, and if the value is small, then only a small number of pages can be printed at full upgrade capability, and power must stay on while doing so.

3.5.4   Use of a PrintEngineLicense id

Silverbrook O/S program code contains the OEM's asymmetric public key to ensure that the subsequent OEM program code is authentic - i.e. from the OEM. However given that SoPEC only contains a single root key, it is theoretically possible for different OEM's applications to be run identically *physical* Print Engines i.e. printer driver for OEM_1 run on an identically *physical* Print Engine from OEM_2.

To guard against this, the Silverbrook O/S program code contains a *PrintEngineLicense_id* code (e.g. 16 bits) that matches the same named value stored as a fixed operating parameter in the PRINTER_QA (i.e. in M_{1+}). As with all other operating parameters, the value of *PrintEngineLicense_id* would be stored in PRINTER_QA at the

same time as the other various PRINTER_QA customizations are being applied, before being shipped to the OEM site.

In this way, the OEMs can be sure of differentiating themselves through software functionality.

3.5.5   Authentication of Ink

The Silverbrook O/S must perform ink authentication [5] during prints. Ink usage authentication makes use of counters in SoPEC that keep an accurate record of the exact number of dots printed for each ink.

The ink amount remaining in a given cartridge is stored in that cartridge's INK_QA chip. Other data stored on the INK_QA chip includes ink color, viscosity, Memjet firing pulse profile information, as well as licensing parameters such as OEM_Id, inkType, InkUsageLicense_Id, etc. This information is typically constant, and is therefore likely to be stored in M_{1+} within INK_QA.

Just as the Print Engine operating parameters are validated by means of PRINTER_QA, a given Print Engine license may only be permitted to function with specifically licensed ink. Therefore the software on SoPEC could contain a valid set of ink types, colors, OEM_Ids, InkUsageLicense_Ids etc. for subsequent matching against the data in the INK_QA.

SoPEC must be able to authenticate reads from the INK_QA, both in terms of ink parameters as well as ink remaining.

To authenticate ink a number of steps must be taken:

- restrict access to dot counts
- authenticate ink usage and ink parameters via INK_QA and PRINTER_QA
- broadcast ink dot usage to all SoPECs in a multi-SoPEC system

3.5.5.1   restrict access to dot counts

Since the dot counts are accessed via the PHI in the PEP section of SoPEC, access to these registers (and more generally *all* PEP registers) must be only available from supervisor mode, and not by OEM code (running in user mode). Otherwise it might be possible for OEM program code to clear dot counts before authentication has occurred.

3.5.5.2   authenticate ink usage and ink parameters via INK_QA and PRINTER_QA

The basic problem of authentication of ink remaining and other ink data boils down to the problem that we don't trust INK_QA. Therefore how can a SoPEC know the initial value of ink (or the ink parameters), and how can a SoPEC know that after a write to the INK_QA, the count has been correctly decremented.

Taking the first issue, which is determining the initial ink count or the ink parameters, we need a system whereby a given SoPEC can perform an authenticated read of the data in INK_QA.

We cannot write the *SoPEC_id_key* to the INK_QA for two reasons:

- updating keys is not power-safe (i.e. if power is removed mid-update, the INK_QA could be rendered useless)

- the ink cartridge would then not work in another printer since the other printer would not know the old *SoPEC_id_key* (knowledge of the old key is required in order to change the old key to a new one).

The proposed solution is to let INK_QA have two keys:

- K_0 = *SupplyInkLicense_key*. This key is constant for all ink cartridges for a given ink supply agreement between an OEM and a Silverbrook ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_0 has write permissions to the ink remaining regions of M_0 on INK_QA.
- K_1 = *UseInkLicense_key*. This is key is constant for all ink cartridges for a given ink usage agreement between an OEM and a Silverbrook ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_1 has no write permissions to anything.

K_0 is used to authenticate the actual upgrades of the amount of ink remaining (e.g. to fill and refill the amount of ink). Upgrades are performed using the standard upgrade protocol described in [5], with INK_QA acting as the ChipU, and the external upgrader acting as the ChipS. The fill and refill upgrader (ChipS) also needs to check the appropriate ink licensing parameters such as OEM_Id, InkType and InkUsageLicense_Id for validity.

K_1 is used to allow SoPEC to authenticate reads of the ink remaining and any other ink data. This is accomplished by having the same *UseInkLicense_key* within PRINTER_QA (e.g. in K_2), also with no write permissions.

This means there are two shared keys, with PRINTER_QA sharing both, and thereby acting as a bridge between INK_QA and SoPEC.

- *UseInkLicense_key* is shared between INK_QA and PRINTER_QA
- *SoPEC_id_key* is shared between SoPEC and PRINTER_QA

All SoPEC has to do is do an authenticated read [5] from INK_QA, pass the data / signature to PRINTER_QA, let PRINTER_QA validate the data / signature, and then get PRINTER_QA to produce a similar signature based on the shared *SoPEC_id_key*. SoPEC can then compare PRINTER_QA's signature with its own calculated signature (i.e. implement a Test function [5] in software on the SoPEC), and if the signatures match, the data from INK_QA must be valid, and can therefore be trusted.

Once the data from INK_QA is known to be trusted, the amount of ink remaining can be checked, and the other ink licensing parameters such as OEM_Id, InkType, InkUsageLicense_Id can be checked for validity.

The actual steps of read authentication as performed by SoPEC are:

```
KEY1 ← 1 // simple constants to specify which key to use when signing
KEY2 ← 2
R_PRINTER ← PRINTER_QA.random()
R_INK, M_INK, SIG_INK ← INK_QA.read(KEY1, R_PRINTER)// read with key1: UseInkLicense_key
R_SOPEC ← random()
SIG_PRINTER ← PRINTER_QA.test(KEY2, R_INK, M_INK, SIG_INK, KEY1, R_SOPEC)
SIG_SOPEC ← HMAC_SHA_1(R_PRINTER | R_SOPEC | M_INK)
If ((SIG_PRINTER != 0) AND (SIG_PRINTER = SIG_SOPEC))
    // M_INK (data read from INK_QA) is valid
    // M_INK could be ink parameters, such as InkUsageLicense_Id, or ink remaining
    If (M_INK.inkRemaining = expectedInkRemaining)
        // all is ok
    Else
```

```
            // the ink value is not what we wrote, so don't print anything anymore
        EndIf
   Else
        // the data read from INK_QA is not valid and cannot be trusted
   EndIf
```

Strictly speaking, we don't need a nonce (R_{SOPEC}) all the time because M_A (containing the ink remaining) should be decrementing between authentications. However we do need one to retrieve the initial amount of ink and the other ink parameters (at power up). This is why taking a random number from the *WatchDogTimer* at the receipt of the first page is acceptable.

In summary, the SoPEC performs the non-authenticated write [5] of ink remaining to the INK_QA chip, and then performs an authenticated read of the data via the PRINTER_QA as per the pseudocode above. If the value is authenticated, *and* the INK_QA ink-remaining value matches the expected value, the count was correctly decremented and the printing can continue.

3.5.5.3   broadcast ink dot usage to all SoPECs in a multi-SoPEC system

In a multi-SoPEC system, each SoPEC *attached to a printhead* (4 at most) must broadcast its ink usage to all the SoPECs. In this way, each SoPEC will have its own version of the expected ink usage.

In the case of a man-in-the-middle attack, at worst the count in a given SoPEC is only its own count (i.e. all broadcasts are turned into 0 ink usage by the man-in-the-middle).

A single SoPEC performs the update of ink remaining to the INK_QA chip, and then all SoPECs perform an authenticated read of the data via the appropriate PRINTER_QA (the PRINTER_QA that contains their matching *SoPEC_id_key* - remember that multiple *SoPEC_id_keys* can be stored in a single PRINTER_QA). If the value is authenticated, and the INK_QA value matches the expected value, the count was correctly decremented and the printing can continue.

If any of the broadcasts are not received, or have been tampered with, the updated ink counts will not match. The only case this does not cater for is if each SoPEC is tricked (via an ISI man-in-the-middle attack), into a total that is the same, yet not the true total. Apart from the fact that this is not viable for general pages, at worst this is the maximum amount of ink printed by a single SoPEC. We don't care about protecting against this case.

Since there will be at most 4 printing SoPEC, it requires at most 4 authenticated reads. This should be completed within 0.5 seconds - well within the 2 seconds/page print time.

3.5.6   Example hierarchy

The exact breakdown of hierarchy will depend on a later investigation, but for the purposes of scoping out possibilities, it is worthwhile considering an example hierarchy for illustrative purposes.

Adding an extra bootloader step to the example from Section 3.5.2, we can break up the contents of program space into logical sections, as shown in Table 1. Note that the ComCo does not provide any program code, merely operating parameters that is used by the O/S.

Table 1. Sections of Program Space

Section	Contents	Verifies
0 (ROM)	boot loader 0 SHA-1 function asymmetric decrypt function boot0key	section 1 via boot0key
1	boot loader 1 SoPEC_OS_public_key	section 2 via SoPEC_OS_public_key
2	Silverbrook O/S program code function to generate SoPEC_id_key from SoPEC_id Basic Print Engine ComCo_public_key	section 3 via ComCo_public_key section 4 via OEM_public_key (supplied in section 3) PRINTER_QA data, which includes the PrintEngineLicense_id, Silverbrook operating parameters, and OEM operating parameters (all authenticated via SoPEC_id_key)
3	ComCo license agreement operating parameter ranges, including PrintEngineLicense_id (gets loaded into supervisor mode section of memory) OEM_public_key (gets loaded into supervisor mode section of memory) Any ComCo written user-mode program code (gets loaded into mode mode section of memory)	Is used by section 2 to verify section 4 and range of parameters as found in PRINTER_QA
4	OEM specific program code	OEM operating parameters via calls to Silverbrook O/S code

The verification procedures will be required each time the CPU is woken up, since the RAM is not preserved.

3.5.7   What if the CPU is not fast enough?

In the example of Section 3.5.6, every time the CPU is woken up to print a document it needs to perform:

- SHA-1 on all program code and program data
- 4 sets of asymmetric decryption to load the program code and data
- 1 HMAC-SHA1 generation per 512-bits of Silverbrook and OEM printer and ink operating parameters

Although the SHA-1 and HMAC process will be fast enough on the embedded CPU (the program code will be executing from ROM), it may be that the asymmetric decryption will be slow. And this becomes more likely with each extra level of authentication. If this is the case (as is likely), hardware acceleration is required.

A cheap form of hardware acceleration takes advantage of the fact that in most cases the same program is loaded each time, with the first time likely to be at power-up. The hardware acceleration is simply data storage for the *authorizedDigest* which means that the boot procedure now is:

```
slowCPU_bootloader0(data, sig)
   localDigest ← SHA-1(data)
   If (localDigest = previouslyStoredAuthorizedDigest)
      jump to program code at data-start address// will never to return
   Else
      authorizedDigest ← decrypt(sig, boot0key)
      If (localDigest = authorizedDigest)
         previouslyStoredAuthorizedDigest ← authorizedDigest
         jump to program code at data-start address// will never to return
      Else
      // program code is unauthorized
   EndIf
```

This procedure means that a reboot of the same authorized program code will only require SHA-1 processing. At power-up, or if new program code is loaded (e.g. an upgrade of a driver over the internet), then the full authorization via asymmetric decryption takes place. This is because the stored digest will not match at power-up and whenever a new program is loaded.

The question is how much preserved space is required.

Each digest requires 160 bits (20 bytes), and this is constant regardless of the asymmetric encryption scheme or the key length. While it is possible to reduce this number of bits, thereby sacrificing security, the cost is small enough to warrant keeping the full digest.

However each level of boot loader requires its own digest to be preserved. This gives a maximum of 20 bytes per loader. Digests for operating parameters and ink levels may also be preserved in the same way, although these authentications should be fast enough not to require cached storage.

Assuming SoPEC provides for 12 digests (to be generous), this is a total of 240 bytes. These 240 bytes could easily be stored as 60×32-bit registers, or probably more conveniently as a small amount of RAM (eg 0.25 - 1 Kbyte). Providing something like 1 Kbyte of RAM has the advantage of allowing the CPU to store other useful data, although this is not a requirement.

In general, it is useful for the boot ROM to know whether it is being started up due to power-on reset or activity on the USB/ISI. In the former case, it can ignore the previously stored values (either 0 for registers or garbage for RAM). In the latter case, it can use the previously stored values. Even without this, a startup value of 0 (or garbage) means the digest won't match and therefore the authentication will occur implictly.

3.6   SoPEC ISI IDENTIFICATION

At power-up, the host can send targeted data to the USB-connected SoPEC, but can only send broadcasts to all of the slave SoPECs via the USB-connected SoPEC's ISI.

Each slave SoPEC will verify the broadcast message received over the ISI, and if it is valid, will execute it. Several levels of authorization may occur. However, at some stage, this common program code (broadcast to all of the slave SoPECs and signed by the appropriate asymmetric private key) will, among other things, set the slave SoPEC's ISI id. If there is only 1 slave, the id is given, but if there is more than 1 slave, the id must be determined in some fashion.

On a particular physical arrangement of SoPECs each slave SoPEC will have a different set of connections on GPIOs. For example, one SoPEC maybe in charge of motor control, while another may be driving the LEDs etc. The unused GPIO pins (not necessarily the same on each SoPEC) can be set as inputs and then tied to 0 or 1. As long as the connection settings are mutually exclusive, program code can determine which is which, and the id appropriately set.

In some multi-SoPEC systems, a given SoPEC will only be attached to a single printhead (left or right). We can conveniently use the second printhead connection pins (temperature and test) to form an ISI id.

This scheme of slave SoPEC identification does not introduce a security breach. If an attacker rewires the pinouts to confuse identification, at best it will simply cause strange printouts (e.g. swapping of printout data) to occur, while at worst the Print Engine will simply not function.

Note that some physical setting (e.g. pins) on each of the multiple SoPECs is required - the settings just need to be mutually exclusive. Although it is possible for all the SoPECs to come to a logical ISI id assignment (e.g. by using ethernet-like protocols), the ISI id needs to be very much a *physical* identity scheme. This is because these SoPECs are not simply logical processors - we want the correct portion of the page to be printed on the correct physical location, motor controls will be physically connected to a specific physical SoPEC etc.

3.7   SETTING UP QA CHIP KEYS

In use, each INK_QA chip needs the following keys:

- K_0 = *SupplyInkLicense_key*
- K_1 = *UseInkLicense_key*

Each PRINTER_QA chip tied to a specific SoPEC requires the following keys:

- K_0 = *PrintEngineLicense_key*
- K_1 = *SoPEC_id_key*
- K_2 = *UseInkLicense_key*

Note that there may be more than one K_1 depending on the number of PRINTER_QA chips and SoPECs in a system. These keys need to be appropriately set up in the QA Chips before they will function correctly together.

3.7.1   Original QA Chips as received by a ComCo

When original QA Chips are shipped from QACo to a specific ComCo their keys are as follows:

- K_0 = *QACo_ComCo_Key0*
- K_1 = *QACo_ComCo_Key1*
- K_2 = *QACo_ComCo_Key2*
- K_3 = *QACo_ComCo_Key3*

All 4 keys are only known to QACo. Note that these keys are different for each QA Chip.

3.7.2   Steps at the ComCo

The ComCo is responsible for making Print Engines out of Memjet printheads, QA Chips, PECs or SoPECs, PCBs etc.

In addition, the ComCo must customize the INK_QA chips and PRINTER_QA chip on-board the print engine before shipping to the OEM.

There are two stages:

- replacing the keys in QA Chips with specific keys for the application (i.e. INK_QA and PRINTER_QA)
- setting operating parameters as per the license with the OEM

3.7.2.1 Replacing keys

The ComCo is issued QID hardware [4] by QACo that allows programming of the various keys (except for K_1) in a given QA Chip to the final values, following the standard ChipF/ChipP replace key (indirect version) protocol [5]. The indirect version of the protocol allows each *QACo_ComCo_Key* to be different for each SoPEC.

In the case of programming of PRINTER_QA's K_1 to be *SoPEC_id_key*, there is the additional step of transferring an asymmetrically encrypted *SoPEC_id_key* (by the public-key) along with the nonce (R_P) used in the replace key protocol to the device that is functioning as a ChipF. The ChipF must decrypt the *SoPEC_id_key* so it can generate the standard replace key message for PRINTER_QA (functioning as a ChipP in the ChipF/ChipP protocol). The asymmetric key pair held in the ChipF equivalent should be unique to a ComCo (but still known only by QACo) to prevent damage in the case of a compromise.

Note that the various keys installed in the QA Chips (both INK_QA and PRINTER_QA) are only known to the QACo. The OEM only uses QIDs and QACo supplied ChipFs. The replace key protocol [5] allows the programming to occur without compromising the old or new key.

3.7.2.2 Setting operating parameters

There are two sets of operating parameters stored in PRINTER_QA and INK_QA:

- fixed
- upgradable

The fixed operating parameters can be written to by means of a non-authenticated writes [5] to M_{1+} via a QID [4], and permission bits set such that they are ReadOnly.

The upgradable operating parameters can only be written to after the QA Chips have been programmed with the correct keys as per Section 3.7.2.1. Once they contain the correct keys they can be programmed with appropriate operating parameters by means of a QID and an appropriate ChipS (containing matching keys).

S3

SoPEC : Hardware Design

Document : SoPEC_hardware_design

Version : 2.3

Author : Brendan Barry
Kelvin Dunne
Paul Furlong
Declan Staunton
Brendan Walsh
Daire Breathnach
Silverbrook Research

Date : 29 Nov 2002

1 Revision History

Version	Date	Released by	Details
0.1	27 May 2002	Brendan Barry	Initial version released to Silverbrook for document structure review.
0.2	15 July 2002	Brendan Barry	S3 Internal version, for review by SoPEC before major version release. All majors chapter are included.
0.3	24 July 2002	Brendan Barry	S3 pre-release version of document, with updates made following S3 Internal review
1.0	24 July 2002	Brendan Barry	Reviewed version for release to Silverbrook
1.1	15 Aug 2002	Silverbrook	Marked up copy of Silverbrook reviewed document
1.2	10 Sept 2002	Daire Breathnach	DNC section Interim release to Silverbrook
1.3	23 Sept 2002	Brendan Barry	Revision 2 pre-release for first pass Internal review
1.4	25 Sept 2002	Brendan Barry	Revision 2 pre-release for second pass Internal review
2.0	27 Sept 2002	Brendan Barry	Revision 2 release to Silverbrook for review
2.1	1 Nov 2002	Brendan Barry	Revision 2 updates after Silverbrook and S3 review
2.2	8 Nov 2002	Brendan Barry	Release contains updates following S3 Internal reviews and Silverbrook interim review of GPIO,SCB,HCU,DNC.
2.3	29 Nov 2002	Brendan Barry	Release with Incorporated updates to all sections based on Silverbrook review feedback. SCB, CPU and DIU sections require further work.

2 Table of Contents

PRINT SYSTEM
OVERVIEW

3 Introduction

This document describes the SoPEC ASIC (Small office home office Print Engine Controller) suitable for use in price sensitive SoHo printer products. The SoPEC ASIC is intended to be a low cost solution for bi-lithic printhead control, replacing the multichip solutions in larger more professional systems with a single chip. The increased cost competitiveness is achieved by integrating several systems such as a modified PEC1 [1] printing pipeline, CPU control system, peripherals and memory sub-system onto one SoC ASIC, reducing component count and simplifying board design.

This section will give a general introduction to Memjet printing systems, introduce the components that make a bi-lithic printhead system, describe possible system architectures and show how several SoPECs can be used to achieve A3 and A4 duplex printing. The section "SoPEC ASIC" describes the SoC SoPEC ASIC, with subsections describing the CPU, DRAM and Print Engine Pipeline subsystems. Each section gives a detailed description of the blocks used and their operation within the overall print system. The final section describes the bi-lithic printhead construction and associated implications to the system due to its makeup.

Some sections of this document were derived from the Print Engine Controller Hardware Design Specification[1] written by Silverbrook Research.

4 Nomenclature

4.1 BI-LITHIC PRINTHEAD NOTATION

A bi-lithic based printhead is constructed from 2 printhead ICs of varying sizes. The notation M:N is used to express the size relationship of each IC, where M specifies one printhead IC in inches and N specifies the remaining printhead IC in inches.

Section 35 Memjet Printhead contains a description of the bi-lithic printhead and related terminology.

4.2 DEFINITIONS

The following terms are used throughout this specification:

Bi-lithic printhead	Refers to printhead constructed from 2 printhead ICs
CPU	Refers to CPU core, caching system and MMU.
ISI-Bridge chip	A device with a high speed interface (such as USB2.0, Ethernet or IEEE1394) and one or more ISI interfaces. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.
ISIMaster	The ISIMaster is the only device allowed to initiate communication on the Inter Sopec Interface (ISI) bus. The ISIMaster interfaces directly with the host.
ISISlave	Multi-SoPEC systems will contain one or more ISISlave SoPECs connected to the ISI bus. ISISlaves can only respond to communication initiated by the ISIMaster.
LEON	Refers to the LEON CPU core.
LineSyncMaster	The LineSyncMaster device generates the line synchronisation pulse that all SoPECs in the system must synchronise their line outputs to.
Multi-SoPEC	Refers to SoPEC based print system with multiple SoPEC devices
Netpage	Refers to page printed with tags (normally in infrared ink).
PEC1	Refers to Print Engine Controller version 1, precursor to SoPEC used to control printheads constructed from multiple angled printhead segments.
Printhead IC	Single MEMS IC used to construct bi-lithic printhead
PrintMaster	The PrintMaster device is responsible for coordinating all aspects of the print operation. There may only be one PrintMaster in a system.
QA Chip	Quality Assurance Chip
Storage SoPEC	An ISISlave SoPEC used as a DRAM store and which does not print.
Tag	Refers to pattern which encodes information about its position and orientation which allow it to be optically located and its data contents read.

4.3 ACRONYM AND ABBREVIATIONS

The following acronyms and abbreviations are used in this specification

CFU	Contone FIFO Unit
CPU	Central Processing Unit
DIU	DRAM Interface Unit

DNC	Dead Nozzle Compensator
DRAM	Dynamic Random Access Memory
DWU	DotLine Writer Unit
GPIO	General Purpose Input Output
HCU	Halftoner Compositor Unit
ICU	Interrupt Controller Unit
ISI	Inter SoPEC Interface
LDB	Lossless Bi-level Decoder
LLU	Line Loader Unit
LSS	Low Speed Serial interface
MEMS	Micro Electro Mechanical System
MMU	Memory Management Unit
PCU	SoPEC Controller Unit
PHI	PrintHead Interface
PSS	Power Save Storage Unit
RDU	Real-time Debug Unit
ROM	Read Only Memory
SCB	Serial Communication Block
SFU	Spot FIFO Unit
SMG4	Silverbrook Modified Group 4.
SoPEC	Small office home office Print Engine Controller
SRAM	Static Random Access Memory
TE	Tag Encoder
TFU	Tag FIFO Unit
TIM	Timers Unit
USB	Universal Serial Bus

4.4 PSEUDOCODE NOTATION

In general the pseudocode examples use C like statements with some exceptions.

Symbol and naming convections used for pseudocode.

//	Comment
=	Assignment
==,!=,<,>	Operator equal, not equal, less than, greater than
+,-,*,/,%	Operator addition, subtraction, multiply, divide, modulus
&,|,^,<<,>>,~	Bitwise AND, bitwise OR, bitwise exclusive OR, left shift, right shift, complement
AND,OR,NOT	Logical AND, Logical OR, Logical inversion
[XX:YY]	Array/vector specifier

| {a, b, c} | Concatenation operation |
| ++, -- | Increment and decrement |

4.4.1 Register and signal naming conventions

In general register naming uses the C style conventions with capitalization to denote word delimiters. Signals use RTL style notation where underscore denote word delimiters. There is a direct translation between both convention. For example the *CmdSourceFifo* register is equivalent to *cmd_source_fifo* signal.

4.5 STATE MACHINE NOTATION

State machines should be described using the pseudocode notation outlined above. State machine descriptions use the convention of underline to indicate the cause of a transition from one state to another and plain text (no underline) to indicate the effect of the transition i.e. signal transitions which occur when the new state is entered.

A sample state machine is shown in Figure 1.
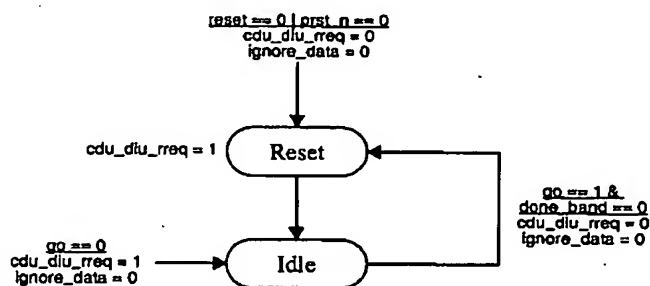
Figure 1. Example State machine notation

5   Printing Considerations

A bi-lithic printhead produces 1600 dpi bi-level dots. On low-diffusion paper, each ejected drop forms a 22.5μm diameter dot. Dots are easily produced in isolation, allowing dispersed-dot dithering to be exploited to its fullest. Since the bi-lithic printhead is the width of the page and operates with a constant paper velocity, color planes are printed in perfect registration, allowing ideal dot-on-dot printing. Dot-on-dot printing minimizes 'muddying' of midtones caused by inter-color bleed.

A page layout may contain a mixture of images, graphics and text. Continuous-tone (contone) images and graphics are reproduced using a stochastic dispersed-dot dither. Unlike a clustered-dot (or amplitude-modulated) dither, a *dispersed-dot* (or frequency-modulated) dither reproduces high spatial frequencies (i.e. image detail) almost to the limits of the dot resolution, while simultaneously reproducing lower spatial frequencies to their full color depth, when spatially integrated by the eye. A *stochastic* dither matrix is carefully designed to be free of objectionable low-frequency patterns when tiled across the image. As such its size typically exceeds the minimum size required to support a particular number of intensity levels (e.g. 16×16× 8 bits for 257 intensity levels).

Human contrast sensitivity peaks at a spatial frequency of about 3 cycles per degree of visual field and then falls off logarithmically, decreasing by a factor of 100 beyond about 40 cycles per degree and becoming immeasurable beyond 60 cycles per degree [21][22]. At a normal viewing distance of 12 inches (about 300mm), this translates roughly to 200-300 cycles per inch (cpi) on the printed page, or 400-600 samples per inch according to Nyquist's theorem.

In practice, contone resolution above about 300 ppi is of limited utility outside special applications such as medical imaging. Offset printing of magazines, for example, uses contone resolutions in the range 150 to 300 ppi. Higher resolutions contribute slightly to color error through the dither.

Black text and graphics are reproduced directly using bi-level black dots, and are therefore not anti-aliased (i.e. low-pass filtered) before being printed. Text should therefore be *supersampled* beyond the perceptual limits discussed above, to produce smoother edges when spatially integrated by the eye. Text resolution up to about 1200 dpi continues to contribute to perceived text sharpness (assuming low-diffusion paper, of course).

A Netpage printer, for example, may use a contone resolution of 267 ppi (i.e. 1600 dpi / 6), and a black text and graphics resolution of 800 dpi. A high end office or departmental printer may use a contone resolution of 320 ppi (1600 dpi / 5) and a black text and graphics resolution of 1600 dpi. Both formats are capable of exceeding the quality of commercial (offset) printing and photographic reproduction.

6   Document Data Flow

6.1   CONSIDERATIONS

Because of the page-width nature of the bi-lithic printhead, each page must be printed at a constant speed to avoid creating visible artifacts. This means that the printing speed can't be varied to match the input data rate. Document rasterization and document printing are therefore decoupled to ensure the printhead has a constant supply of data. A page is never printed until it is fully rasterized. This can be achieved by storing a compressed version of each rasterized page image in memory.

This decoupling also allows the RIP(s) to run ahead of the printer when rasterizing simple pages, buying time to rasterize more complex pages.

Because contone color images are reproduced by stochastic dithering, but black text and line graphics are reproduced directly using dots, the compressed page image format contains a separate foreground bi-level black layer and background contone color layer. The black layer is composited over the contone layer after the contone layer is dithered (although the contone layer has an optional black component). A final layer of Netpage tags (in infrared or black ink) is optionally added to the page for printout.

Figure 2 shows the flow of a document from computer system to printed page.
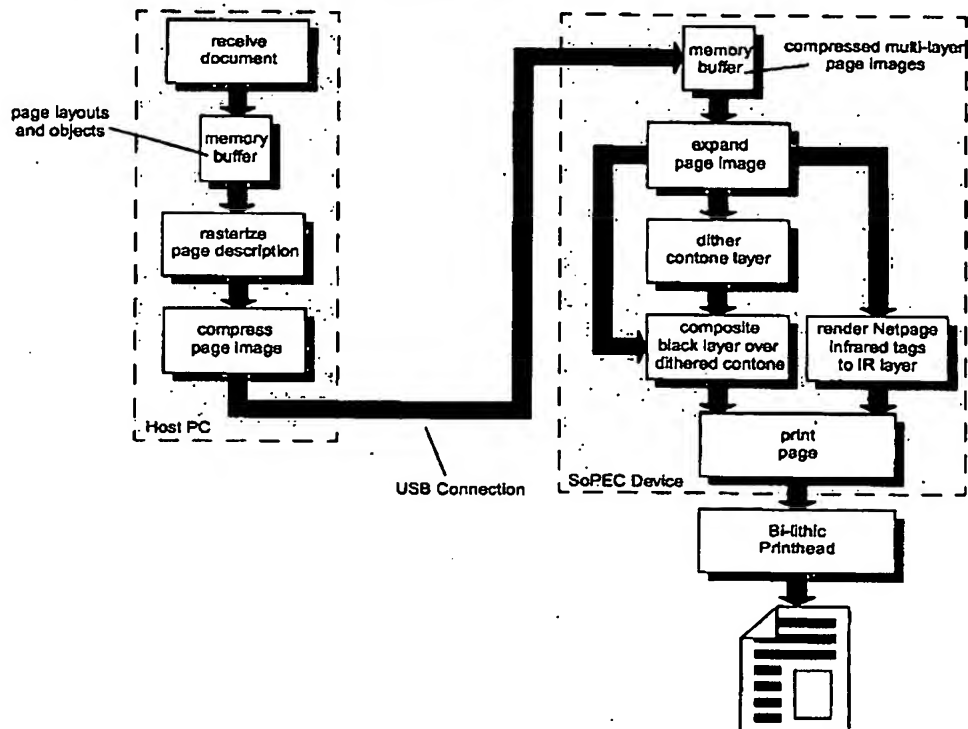
Figure 2. Document data flow

At 267 ppi for example, a A4 page (8.26 inches × 11.7 inches) of contone CMYK data has a size of 26.3MB. At 320 ppi, an A4 page of contone data has a size of 37.8MB. Using lossy contone compression algorithms such as JPEG [23], contone images compress with a ratio up to 10:1 without noticeable loss of quality, giving compressed page sizes of 2.63MB at 267 ppi and 3.78 MB at 320 ppi.

At 800 dpi, a A4 page of bi-level data has a size of 7.4MB. At 1600 dpi, a Letter page of bi-level data has a size of 29.5 MB. Coherent data such as text compresses very well. Using lossless bi-level compression algorithms such as SMG4 fax as discussed in Section 8.1.2.3.1, ten-point plain text compresses with a ratio of about 50:1. Lossless bi-level compression across an average page is about 20:1 with 10:1 possible for pages which compress poorly. The requirement for SoPEC is to be able to print text at 10:1 compression. Assuming 10:1 compression gives compressed page sizes of 0.74 MB at 800 dpi, and 2.95 MB at 1600 dpi.

Once dithered, a page of CMYK contone image data consists of 116MB of bi-level data. Using lossless bi-level compression algorithms on this data is pointless precisely because the optimal dither is stochastic - i.e. since it introduces hard-to-compress disorder.

Netpage tag data is optionally supplied with the page image. Rather than storing a compressed bi-level data layer for the Netpage tags, the tag data is stored in its raw form. Each tag is supplied up to 120 bits of raw variable data (combined with up to 56 bits of raw fixed data) and covers up to a 6mm × 6mm area (at 1600 dpi). The absolute maximum number of tags on a A4 page is 15,540 when the tag is only 2mm × 2mm (each tag is 126 dots × 126 dots, for a total coverage of 148 tags × 105 tags). 15,540 tags of 128 bits per tag gives a compressed tag page size of 0.24 MB.

The multi-layer compressed page image format therefore exploits the relative strengths of lossy JPEG contone image compression, lossless bi-level text compression, and tag encoding. The format is compact enough to be storage-efficient, and simple enough to allow straightforward real-time expansion during printing.

Since text and images normally don't overlap, the normal worst-case page image size is image only, while the normal best-case page image size is text only. The addition of worst case Netpage tags adds 0.24MB to the page image size. The worst-case page image size is text over image plus tags. The average page size assumes a quarter of an average page contains images. Table 1 shows data sizes for compressed Letter page for these different options.

Table 1. Data sizes for A4 page (8.26 inches × 11.7 inches)

	267 ppi contone 800 dpi bi-level	320 ppi contone 1600 dpi bi-level
Image only (contone), 10:1 compression	2.63 MB	3.78 MB
Text only (bi-level), 10:1 compression	0.74 MB	2.95 MB
Netpage tags, 1600 dpi	0.24 MB	0.24 MB
Worst case (text + image + tags)	3.61 MB	6.67 MB
Average (text + 25% image + tags)	1.64 MB	4.25 MB

6.2 DOCUMENT DATA FLOW

The Host PC rasterizes and compresses the incoming document on a page by page basis. The page is restructured into bands with one or more bands used to construct a page. The compressed data is then transferred to the SoPEC device via the USB link. A complete band is stored in SoPEC embedded memory. Once the band transfer is complete the SoPEC device reads the compressed data, expands the band, normalizes contone, bi-level and tag data to 1600 dpi and transfers the resultant calculated dots to the bilithic printhead.

The document data flow is

- The RIP software rasterizes each page description and compress the rasterized page image.
- The infrared layer of the printed page optionally contains encoded Netpage [5] tags at a programmable density.
- The compressed page image is transferred to the SoPEC device via the USB normally on a band by band basis.
- The print engine takes the compressed page image and starts the page expansion.
- The first stage page expansion consists of 3 operations performed in parallel
  - expansion of the JPEG-compressed contone layer
  - expansion of the SMG4 fax compressed bi-level layer
  - encoding and rendering of the bi-level tag data.
- The second stage dithers the contone layer using a programmable dither matrix, producing up to four bi-level layers at full-resolution.
- The second stage then composites the bi-level tag data layer, the bi-level SMG4 fax de-compressed layer and up to four bi-level JPEG de-compressed layers into the full-resolution page image.
- A fixative layer is also generated as required.
- The last stage formats and prints the bi-level data through the bi-lithic printhead via the printhead interface.

The SoPEC device can print a full resolution page with 6 color planes. Each of the color planes can be generated from compressed data through any channel (either JPEG compressed, bi-level SMG4 fax compressed, tag data generated, or fixative channel created) with a maximum number of 6 data channels from page RIP to bi-lithic printhead color planes.

The mapping of data channels to color planes is programmable, this allows for multiple color planes in the printhead to map to the same data channel to provide for redundancy in the printhead to assist dead nozzle compensation.

Also a data channel could be used to gate data from another data channel. For example in stencil mode, data from the bilevel data channel at 1600 dpi can be used to filter the contone data channel at 320 dpi, giving the effect of 1600 dpi contone image.

6.3   PAGE CONSIDERATIONS DUE TO SoPEC

The SoPEC device typically stores a complete page of document data on chip. The amount of storage available for compressed pages is limited to 2Mbytes, imposing a fixed maximum on compressed page size. A comparison of the compressed image sizes in Table 1 indicates that SoPEC would not be capable of printing worst case pages unless they are split into bands and printing commences before all the bands for the page have been downloaded. The page sizes in the table are shown for comparison purposes and would be considered reasonable for a professional level printing system. The SoPEC device is aimed at the consumer level and would not be required to print pages of that complexity. Target document types for the SoPEC device are shown Table 2.

Table 2. Page content targets for SoPEC

Page Content Description	Calculation	Size (MByte)
Best Case picture Image, 267ppi with 3 colors, A4 size	8.26x11.7x267x267x3 @ 10:1	1.97
Full page text, 800dpi A4 size	8.26x11.7x800x800 @ 10:1	0.74

Table 2. Page content targets for SoPEC

Page Content Description	Calculation	Size (MByte)
Mixed Graphics and Text - Image of 6 inches x 4 inches @ 267 ppi and 3 colors - Remaining area text ~73 inches^2, 800 dpi	6x4x267x267x3 @ 5:1 800x800x73 @ 10:1	1.55
Best Case Photo, 3 Colors, 6.6 MegaPixel Image	6.6 Mpixel @ 10:1	2.00

If a document with more complex pages is required, the page RIP software in the host PC can determine that there is insufficient memory storage in the SoPEC for that document. In such cases the RIP software can take two courses of action. It can increase the compression ratio until the compressed page size will fit in the SoPEC device, at the expense of document quality, or divide the page into bands and allow SoPEC to begin printing a page band before all bands for that page are downloaded. Once SoPEC starts printing a page it cannot stop, if SoPEC consumes compressed data faster than the bands can be downloaded a buffer underrun error could occur causing the print to fail. A buffer underrun occurs if line synchronisation pulse is received before a line of data has been transferred to the printhead.

Other options which can be considered if the page does not fit completely into the compressed page store are to slow the printing or to use multiple SoPECs to print parts of the page. A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

7 · Memjet Printer Architecture

The SoPEC device can be used in several printer configurations and architectures.

In the general sense every SoPEC based printer architecture will contain:
- One or more SoPEC devices.
- One or more bi-lithic printheads.
- Two or more LSS busses.
- Two or more QA chips.
- USB 1.1 connection to host or ISI connection to Bridge Chip.
- ISI bus connection between SoPECs (when multiple SoPECs are used).

Some example printer configurations as outlined in Section 7.2. The various system components are outlined briefly in Section 7.1.

7.1 SYSTEM COMPONENTS

7.1.1 SoPEC Print Engine Controller

The SoPEC device contains several system on a chip (SoC) components, as well as the print engine pipeline control application specific logic.

7.1.1.1 Print Engine Pipeline (PEP) Logic

The PEP reads compressed page store data from the embedded memory, optionally decompresses the data and formats it for sending to the printhead. The print engine pipeline functionality includes expanding the page image, dithering the contone layer, compositing the black layer over the contone layer, rendering of Netpage tags, compensation for dead nozzles in the printhead, and sending the resultant image to the bi-lithic printhead.

7.1.1.2 Embedded CPU

SoPEC contains an embedded CPU for general purpose system configuration and management. The CPU performs page and band header processing, motor control and sensor monitoring (via the GPIO) and other system control functions. The CPU can perform buffer management or report buffer status to the host. The CPU can optionally run vendor application specific code for general print control such as paper ready monitoring and LED status update.

7.1.1.3 Embedded Memory Buffer

A 2.5Mbyte embedded memory buffer is integrated onto the SoPEC device, of which approximately 2Mbytes are available for compressed page store data. A compressed page is divided into one or more bands, with a number of bands stored in memory. As a band of the page is consumed by the PEP for printing a new band can be downloaded. The new band may be for the current page or the next page.

Using banding it is possible to begin printing a page before the complete compressed page is downloaded, but care must be taken to ensure that data is always available for printing or a buffer underrun may occur.

An Storage SoPEC acting as a memory buffer (Section 7.2.5) or an ISI-Bridge chip with attached DRAM (Section 7.2.6) could be used to provide guaranteed data delivery.

7.1.1.4 Embedded USB 1.1 Device

The embedded USB 1.1 device accepts compressed page data and control commands from the host PC, and facilitates the data transfer to either embedded memory or to another SoPEC device in multi-SoPEC systems.

7.1.2 Bi-lithic Printhead

The printhead is constructed by abutting 2 printhead ICs together. The printhead ICs can vary in size from 2 inches to 8 inches, so to produce an A4 printhead several combinations are possible. For example two printhead ICs of 7 inches and 3 inches could be used to create a A4 printhead (the notation is 7:3). Similarly 6 and 4 combination (6:4), or 5:5 combination. For an A3 printhead it can be constructed from 8:6 or an 7:7 printhead IC combination. For photographic printing smaller printheads can be constructed.

7.1.3 LSS interface bus

Each SoPEC device has 2 LSS system buses for communication with QA devices for system authentication and ink usage accounting. The number of QA devices per bus and their position in the system is unrestricted with the exception that *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses.

7.1.4 QA devices

Each SoPEC system can have several QA devices. Normally each printing SoPEC will have an associated *PRINTER_QA*. Ink cartridges will contain an *INK_QA* chip. *PRINTER_QA* and *INK_QA* devices should be on separate LSS busses. All QA chips in the system are physically identical with flash memory contents defining *PRINTER_QA* from *INK_QA* chip.

7.1.5 ISI interface

The Inter-SoPEC Interface (ISI) provides a communication channel between SoPECs in a multi-SoPEC system. The ISIMaster can be SoPEC device or an ISI-Bridge chip depending on the printer configuration. Both compressed data and control commands are transferred via the interface.

7.1.6 ISI-Bridge Chip

A device, other than a SoPEC with a USB connection, which provides print data to a number of slave SoPECs. A bridge chip will typically have a high bandwidth connection, such as USB2.0, Ethernet or IEEE1394, to a host and may have an attached external DRAM for compressed page storage. A bridge chip would have one or more ISI interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

7.2 POSSIBLE SoPEC SYSTEMS

Several possible SoPEC based system architectures exist. The following sections outline some possible architectures. It is possible to have extra SoPEC devices in the system used for DRAM storage. The QA chip configurations shown are indicative of the flexibility of LSS bus architecture, but not limited to those configurations.

7.2.1 A4 Simplex with 1 SoPEC device
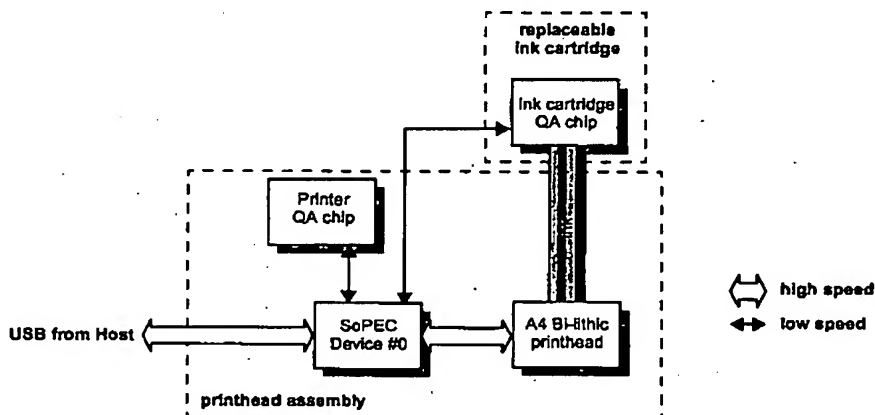
Figure 3. Single SoPEC A4 Simplex system

In Figure 3, a single SoPEC device can be used to control two printhead ICs. The SoPEC receives compressed data through the USB device from the host. The compressed data is processed and transferred to the printhead.

7.2.2 A4 Duplex with 2 SoPEC devices
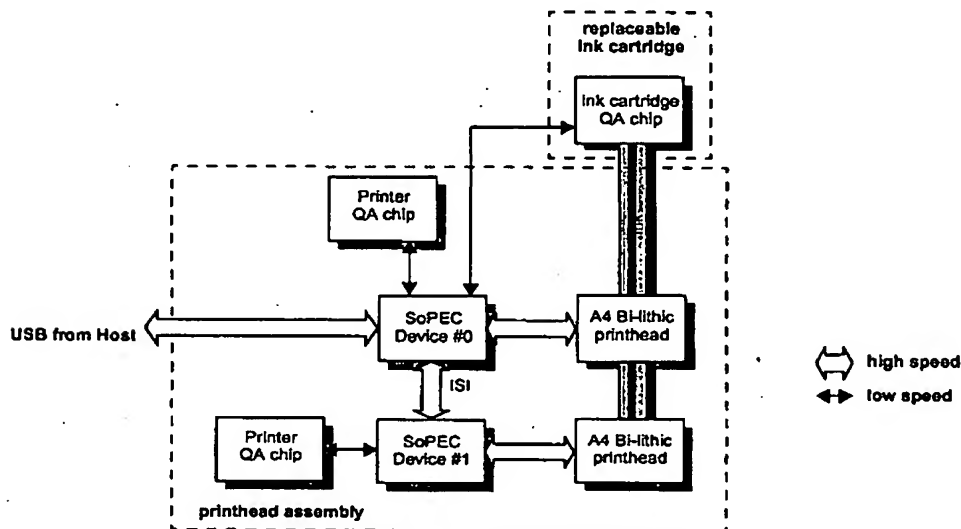
Figure 4. Dual SoPEC A4 Duplex system

In Figure 4, two SoPEC devices are used to control two bi-lithic printheads, each with two printhead ICs. Each bi-lithic printhead prints to opposite sides of the same page to achieve duplex printing. The SoPEC connected to the USB is the ISIMaster SoPEC, the remaining SoPEC is an ISISlave. The ISIMaster receives all the compressed page data for both SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A4 page every 2 seconds in this configuration since the USB 1.1 connection to the host may not have enough bandwidth. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.3 A3 Simplex with 2 SoPEC devices
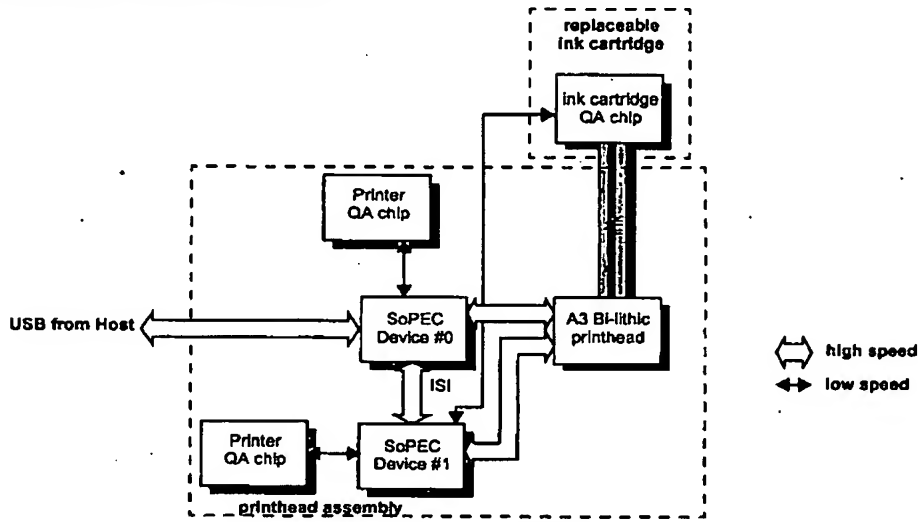
Figure 5. Dual SoPEC A3 simplex system

In Figure 5, two SoPEC devices are used to control one A3 bi-lithic printhead. Each SoPEC controls only one printhead IC (the remaining PHI port typically remains idle). The USB 1.1 connection defines the ISI-Master SoPEC. In this dual SoPEC configuration the compressed page store data is split across 2 SoPECs giving a total of 4Mbyte page store, this allows the system to use compression rates as in an A4 architecture, but with the increased page size of A3. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print slower. An alternative would be for each SoPEC to have its own USB 1.1 connection. This would allow a faster average print speed.

7.2.4   A3 Duplex with 4 SoPEC devices
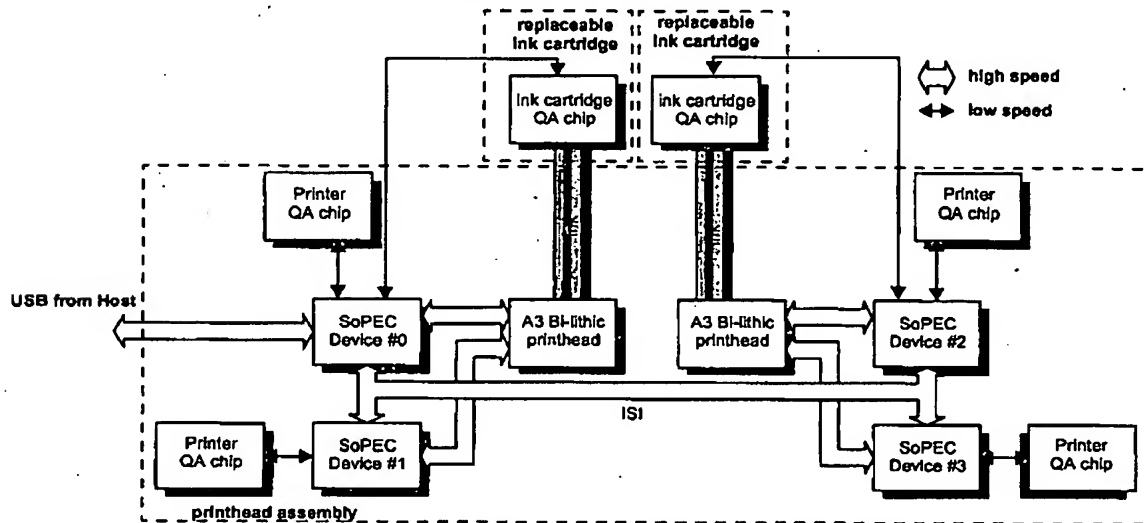
Figure 6. Quad SoPEC A3 duplex system

In Figure 6 a 4 SoPEC system is shown. It contains 2 A3 bi-lithic printheads, one for each side of an A3 page. Each printhead contain 2 printhead ICs, each printhead IC is controlled by an independent SoPEC device, with the remaining PHI port typically unused. Again the USB 1.1 connection defines the ISIMaster with the other SoPECs as ISISlaves. In total, the system contains 8Mbytes of compressed page store (2Mbytes per SoPEC), so the increased page size does not degrade the system print quality, from that of an A4 simplex printer. The ISIMaster receives all the compressed page data for all SoPECs and re-distributes the compressed data over the Inter-SoPEC Interface (ISI) bus.

It may not be possible to print an A3 page every 2 seconds in this configuration since the USB 1.1 connection to the host will only have enough bandwidth to supply 2Mbytes every 2 seconds. Pages which require more than 2MBytes every 2 seconds will therefore print slower. An alternative would be for each SoPEC or set of SoPECS on the same side of the page to have their own USB 1.1 connection. This would allow a faster average print speed.

7.2.5   SoPEC DRAM storage solution: A4 Simplex with 1 printing SoPEC and 1 memory SoPEC
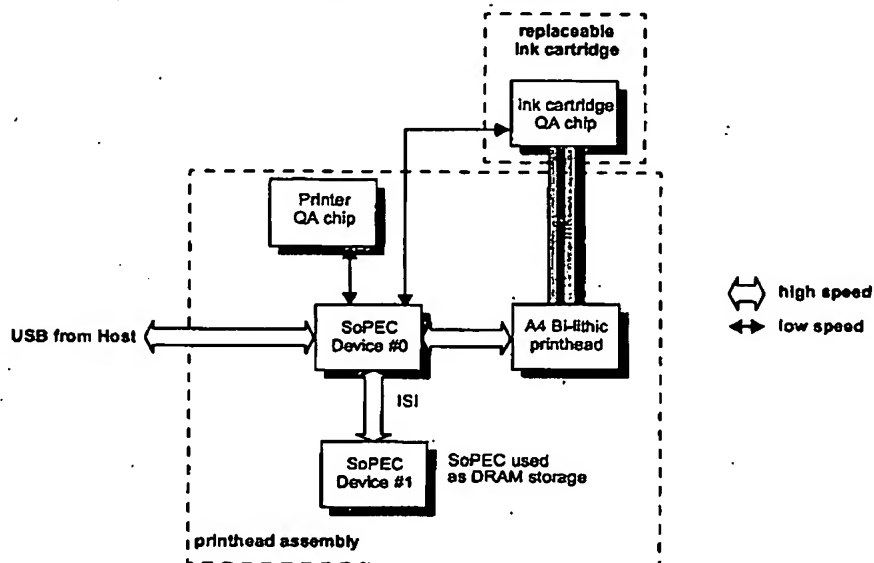
Figure 7.   SoPEC A4 Simplex system with extra SoPEC used as DRAM storage

Extra SoPECs can be used for DRAM storage e.g. in Figure 7 an A4 simplex printer can be built with a single extra SoPEC used for DRAM storage. The DRAM SoPEC can provide guaranteed bandwidth delivery of data to the printing SoPEC. SoPEC configurations can have multiple extra SoPECs used for DRAM storage.

7.2.6 ISI-Bridge chip solution: A3 Duplex system with 4 SoPEC devices
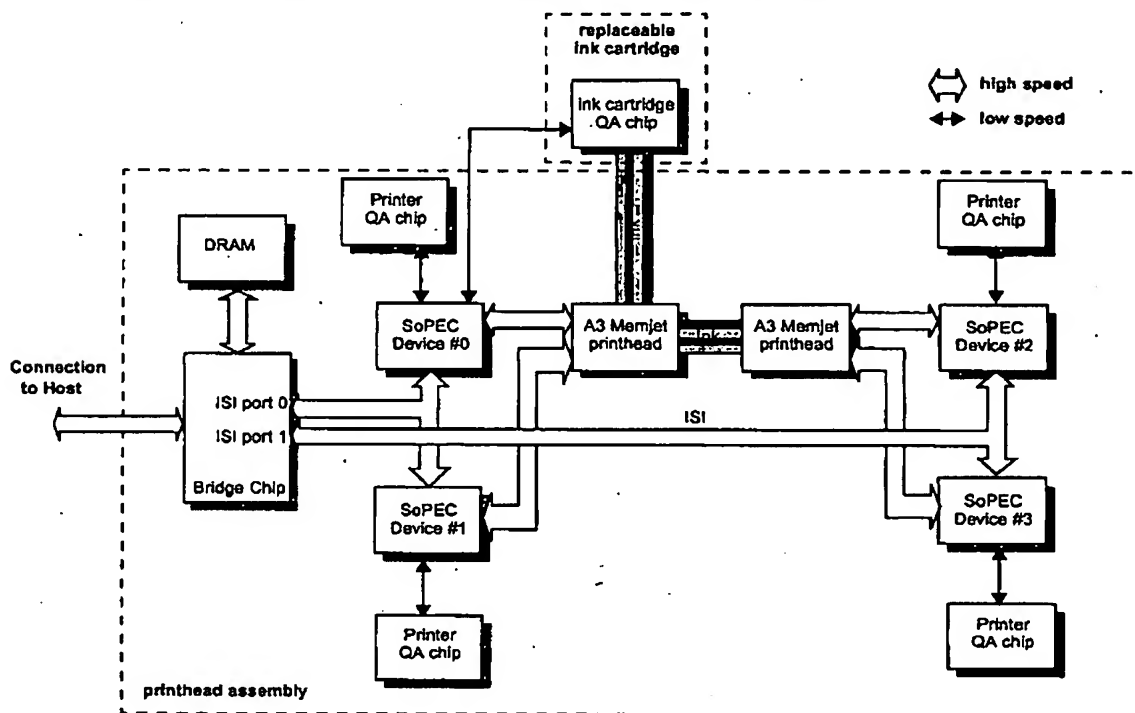
Figure 8. A3 duplex system featuring four printing SoPECs

In Figure 8, an ISI-Bridge chip provides slave-only ISI connections to SoPEC devices. Figure 8 shows a ISI-Bridge chip with 2 separate ISI ports. The ISI-Bridge chip is the ISIMaster on each of the ISI busses it is connected to. All connected SoPECs are ISISlaves. The ISI-Bridge chip will typically have a high bandwidth connection to a host and may have an attached external DRAM for compressed page storage.

An alternative to having a ISI-Bridge chip would be for each SoPEC or each set of SoPECs on the same side of a page to have their own USB 1.1 connection. This would allow a faster average print speed.

8   Page Format and Printflow

When rendering a page, the RIP produces a page header and a number of bands (a non-blank page requires at least one band) for a page. The page header contains high level rendering parameters, and each band contains compressed page data. The size of the band will depend on the memory available to the RIP, the speed of the RIP, and the amount of memory remaining in SoPEC while printing the previous band(s). Figure 9 shows the high level data structure of a number of pages with different numbers of bands in the page.
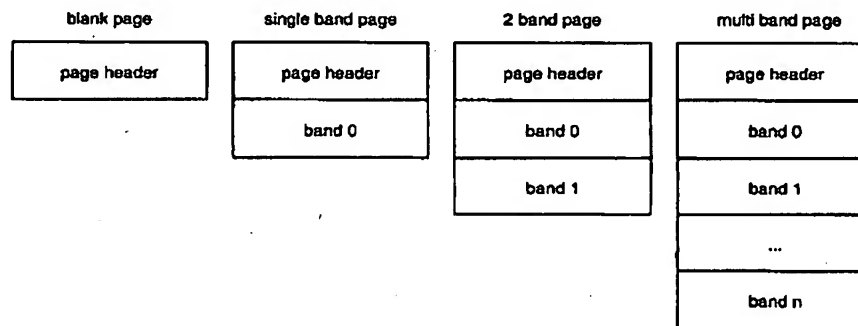
Figure 9. Pages containing different numbers of bands

Each compressed band contains a mandatory band header, an optional bi-level plane, optional sets of interleaved contone planes, and an optional tag data plane (for Netpage enabled applications). Since each of these planes is optional[1], the band header specifies which planes are included with the band. Figure 10 gives a high-level breakdown of the contents of a page band.
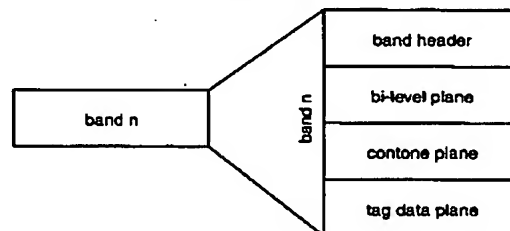
Figure 10. Contents of a page band

A single SoPEC has maximum rendering restrictions as follows:
- 1 bi-level plane
- 1 contone interleaved plane set containing a maximum of 4 contone planes
- 1 tag data plane
- a bi-lithic printhead with a maximum of 2 printhead ICs

The requirement for single-sided A4 single SoPEC printing is
- average contone JPEG compression ratio of 10:1, with a local minimum compression ratio of 5:1 for a single line of interleaved JPEG blocks.

---

1. Although a band must contain at least one plane

- average bi-level compression ratio of 10:1, with a local minimum compression ratio of 1:1 for a single line.

If the page contains rendering parameters that exceed these specifications, then the RIP or the Host PC must split the page into a format that can be handled by a single SoPEC.

In the general case, the SoPEC CPU must analyze the page and band headers and generate an appropriate set of register write commands to configure the units in SoPEC for that page. The various bands are passed to the destination SoPEC(s) to locations in DRAM determined by the host.

The host keeps a memory map for the DRAM, and ensures that as a band is passed to a SoPEC, it is stored in a suitable free area in DRAM. Each SoPEC is connected to the ISI bus or USB bus via its Serial communication Block (SCB). The SoPEC CPU configures the SCB to allow compressed data bands to pass from the USB or ISI through the SCB to SoPEC DRAM. Figure 11 shows an example data flow for a page destined to be printed by a single SoPEC. Band usage information is generated by the individual SoPECs and passed back to the host.
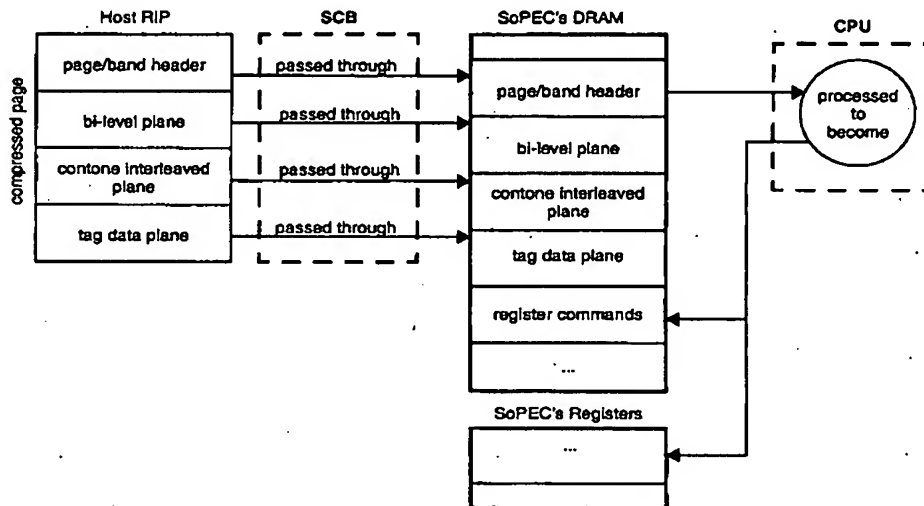
Figure 11. Page data path from host to SoPEC

SoPEC has an addressing mechanism that permits circular band memory allocation, thus facilitating easy memory management. However it is not strictly necessary that all bands be stored together. As long as the appropriate registers in SoPEC are set up for each band, and a given band is contiguous[1], the memory can be allocated in any way.

---

1. Contiguous allocation also includes wrapping around in SoPEC's band store memory.

8.1 PRINT ENGINE EXAMPLE PAGE FORMAT

This section describes a possible format of compressed pages expected by the embedded CPU in SoPEC. The format is generated by software in the host PC and interpreted by embedded software in SoPEC. This section indicates the type of information in a page format structure, but implementations need not be limited to this format. The host PC can optionally perform the majority of the header processing.

The compressed format and the print engines are designed to allow real-time page expansion during printing, to ensure that printing is never interrupted in the middle of a page due to data underrun.

The page format described here is for a single black bi-level layer, a contone layer, and a Netpage tag layer. The black bi-level layer is defined to composite *over* the contone layer.

The black bi-level layer consists of a bitmap containing a 1-bit *opacity* for each pixel. This black layer *matte* has a resolution which is an integer or non-integer factor of the printer's dot resolution. The highest supported resolution is 1600 dpi, i.e. the printer's full dot resolution.

The contone layer, optionally passed in as YCrCb, consists of a 24-bit CMY or 32-bit CMYK *color* for each pixel. This contone image has a resolution which is an integer or non-integer factor of the printer's dot resolution. The requirement for a single SoPEC is to support 1 side per 2 seconds A4/Letter printing at a resolution of 267 ppi, i.e. one-sixth the printer's dot resolution.

Non-integer scaling can be performed on both the contone and bi-level images. Only integer scaling can be performed on the tag data.

The black bi-level layer and the contone layer are both in compressed form for efficient storage in the printer's internal memory.

8.1.1 Page structure

A single SoPEC is able to print with full edge bleed for Letter and A3 via different stitch part combinations of the bi-lithic printhead. It imposes no margins and so has a printable page area which corresponds to the size of its paper. The target page size is constrained by the printable page area, less the explicit (target) left and top margins specified in the page description. These relationships are illustrated below.
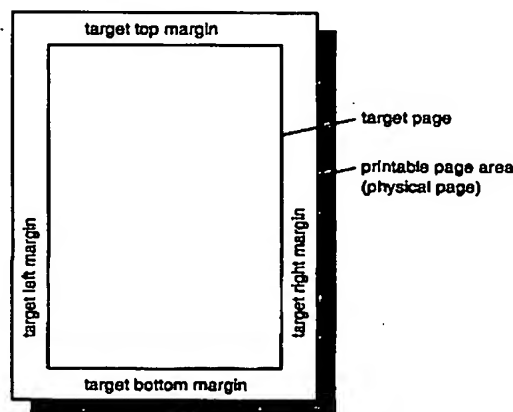
Figure 12. Page structure

8.1.2   Compressed page format

Apart from being implicitly defined in relation to the printable page area, each page description is complete and self-contained. There is no data stored separately from the page description to which the page description refers.[1] The page description consists of a page header which describes the size and resolution of the page, followed by one or more page bands which describe the actual page content.

8.1.2.1   Page header

Table 3 shows an example format of a page header.

Table 3. Page header format

field	format	description
signature	16-bit integer	Page header format signature.
version	16-bit integer	Page header format version number.
structure size	16-bit integer	Size of page header.
band count	16-bit integer	Number of bands specified for this page.
target resolution (dpi)	16-bit integer	Resolution of target page. This is always 1600 for the Memjet printer.
target page width	16-bit integer	Width of target page, in dots.
target page height	32-bit integer	Height of target page, in dots.
target left margin for black and contone	16-bit integer	Width of target left margin, in dots, for black and contone.
target top margin for black and contone	16-bit integer	Height of target top margin, in dots, for black and contone.
target right margin for black and contone	16-bit integer	Width of target right margin, in dots, for black and contone.
target bottom margin for black and contone	16-bit integer	Height of target bottom margin, in dots, for black and contone.
target left margin for tags	16-bit integer	Width of target left margin, in dots, for tags.
target top margin for tags	16-bit integer	Height of target top margin, in dots, for tags.
target right margin for tags	16-bit integer	Width of target right margin, in dots, for tags.
target bottom margin for tags	16-bit integer	Height of target bottom margin, in dots, for tags.
generate tags	16-bit integer	Specifies whether to generate tags for this page (0 - no, 1 - yes).
fixed tag data	128-bit integer	This is only valid if generate tags is set.
tag vertical scale factor	16-bit integer	Scale factor in vertical direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only
tag horizontal scale factor	16-bit integer	Scale factor in horizontal direction from tag data resolution to target resolution. Valid range = 1-511. Integer scaling only.
bi-level layer vertical scale factor	16-bit integer	Scale factor in vertical direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.

---

1. SoPEC relies on dither matrices and tag structures to have already been set up, but these are not considered to be part of a general page format. It is trivial to extend the page format to allow exact specification of dither matrices and tag structures.

Table 3. Page header format

field	format	description
bi-level layer horizontal scale factor	16-bit integer	Scale factor in horizontal direction from bi-level resolution to target resolution (must be 1 or greater). May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
bi-level layer page width	16-bit integer	Width of bi-level layer page, in pixels.
bi-level layer page height	32-bit integer	Height of bi-level layer page, in pixels.
contone flags	16 bit integer	Defines the color conversion that is required for the JPEG data. Bits 2-0 specify how many contone planes there are (e.g. 3 for CMY and 4 for CMYK). Bit 3 specifies whether the first 3 color planes need to be converted back from YCrCb to CMY. Only valid if b2-0 = 3 or 4. 0 - no conversion, leave JPEG colors alone 1 - color convert. Bits 7-4 specifies whether the YCrCb was generated directly from CMY, or whether it was converted to RGB first via the step: R = 255-C, G = 255-M, B = 255-Y. Each of the color planes can be individually inverted. Bit 4: 0 - do not invert color plane 0 1 - invert color plane 0 Bit 5: 0 - do not invert color plane 1 1 - invert color plane 1 Bit 6: 0 - do not invert color plane 2 1 - invert color plane 2 Bit 7: 0 - do not invert color plane 3 1 - invert color plane 3 Bit 8 specifies whether the contone data is JPEG compressed or non-compressed: 0 - JPEG compressed 1 - non-compressed The remaining bits are reserved (0).
contone vertical scale factor	16-bit integer	Scale factor in vertical direction from contone channel resolution to target resolution. Valid range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone horizontal scale factor	16-bit integer	Scale factor in horizontal direction from contone channel resolution to target resolution. Valid range = 1-255. May be non-integer. Expressed as a fraction with upper 8-bits the numerator and the lower 8 bits the denominator.
contone page width	16-bit integer	Width of contone page, in contone pixels.
contone page height	32-bit integer	Height of contone page, in contone pixels.
reserved	up to 128 bytes	Reserved and 0 pads out page header to multiple of 128 bytes.

The page header contains a signature and version which allow the CPU to identify the page header format. If the signature and/or version are missing or incompatible with the CPU, then the CPU can reject the page.

The contone flags define how many contone layers are present, which typically is used for defining whether the contone layer is CMY or CMYK. Additionally, if the color planes are CMY, they can be optionally stored as YCrCb, and further optionally color space converted from CMY directly or via RGB. Finally the contone data is specified as being either JPEG compressed or non-compressed.

The page header defines the resolution and size of the target page. The bi-level and contone layers are clipped to the target page if necessary. This happens whenever the bi-level or contone scale factors are not factors of the target page width or height.

The target left, top, right and bottom margins define the positioning of the target page within the printable page area.

The tag parameters specify whether or not Netpage tags should be produced for this page and what orientation the tags should be produced at (landscape or portrait mode). The fixed tag data is also provided.

The contone, bi-level and tag layer parameters define the page size and the scale factors.

8.1.2.2 Band format

Table 4 shows the format of the page band header.

Table 4. Band header format

field	format	description
signature	16-bit integer	Page band header format signature.
version	16-bit integer	Page band header format version number.
structure size	16-bit integer	Size of page band header.
bi-level layer band height	16-bit integer	Height of bi-level layer band, in black pixels.
bi-level layer band data size	32-bit integer	Size of bi-level layer band data, in bytes.
contone band height	16-bit integer	Height of contone band, in contone pixels.
contone band data size	32-bit integer	Size of contone plane band data, in bytes.
tag band height	16-bit integer	Height of tag band, in dots.
tag band data size	32-bit integer	Size of unencoded tag data band, in bytes. Can be 0 which indicates that no tag data is provided.
reserved	up to 128 bytes	Reserved and 0 pads out band header to multiple of 128 bytes.

The bi-level layer parameters define the height of the black band, and the size of its compressed band data. The variable-size black data follows the page band header.

The contone layer parameters define the height of the contone band, and the size of its compressed page data. The variable-size contone data follows the black data.

The tag band data is the set of variable tag data half-lines as required by the tag encoder. The format of the tag data is found in Section 26.5.2. The tag band data follows the contone data.

Table 5 shows the format of the variable-size compressed band data which follows the page band header.

Table 5. Page band data format

field	format	description
black data	Modified G4 facsimile bitstream[1]	Compressed bi-level layer.
contone data	JPEG bytestream	Compressed contone datalayer.
tag data map	Tag data array	Tag data format. See Section 26.5.2.

1. See section 8.1.2.3 on page 31 for note regarding the use of this standard

The start of each variable-size segment of band data should be aligned to a 256-bit DRAM word boundary.

The following sections describe the format of the compressed bi-level layers and the compressed contone layer. section 26.5.1 on page 365 describes the format of the tag data structures.

8.1.2.3 *Bi-level data compression*

The (typically 1600 dpi) black bi-level layer is losslessly compressed using Silverbrook Modified Group 4 (SMG4) compression which is a version of Group 4 Facsimile compression [18] without Huffman and with simplified run length encodings. Typically compression ratios exceed 10:1. The encoding are listed in Table 6 and Table 7

Table 6. Bi-Level group 4 facsimile style compression encodings

	Encoding	Description
same as Group 4 Facsimile	1000	Pass Command: a0 ← b2, skip next two edges
	1	Vertical(0): a0 ← b1, color = !color
	110	Vertical(1): a0 ← b1 + 1, color = !color
	010	Vertical(-1): a0 ← b1 - 1, color = !color
	110000	Vertical(2): a0 ← b1 + 2, color = !color
	010000	Vertical(-2): a0 ← b1 - 2, color = !color
Unique to this implementation	100000	Vertical(3): a0 ← b1 + 3, color = !color
	000000	Vertical(-3): a0 ← b1 - 3, color = !color
	<RL><RL>100	Horizontal: a0 ← a0 + <RL> + <RL>

SMG4 has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a run-length code, followed by pass through. The pass through escape code is a medium length run-length with a run of less than or equal to 31.

Table 7. Run length (RL) encodings

	Encoding	Description
Unique to this implementation	RRRRR1	Short Black Runlength (5 bits)
	RRRRR1	Short White Runlength (5 bits)
	RRRRRRRRR10	Medium Black Runlength (10 bits)
	RRRRRRRR10	Medium White Runlength (8 bits)
	RRRRRRRRRR10	Medium Black Runlength with RRRRRRRRRR <= 31, Enter pass through
	RRRRRRRR10	Medium White Runlength with RRRRRRRR <= 31, Enter pass through
	RRRRRRRRRRRRRRR00	Long Black Runlength (15 bits)
	RRRRRRRRRRRRRRR00	Long White Runlength (15 bits)

Since the compression is a bitstream, the encodings are read right (least significant bit) to left (most significant bit). The run lengths given as RRRR in Table 7 are read in the same way (least significant bit at the right to most significant bit at the left).

Each band of bi-level data is optionally self contained. The first line of each band therefore is based on a 'previous' blank line or the last line of the previous band.

8.1.2.3.1 Group 3 and 4 facsimile compression

The Group 3 Facsimile compression algorithm [18] losslessly compresses bi-level data for transmission over slow and noisy telephone lines. The bi-level data represents scanned black text and graphics on a white background, and the algorithm is tuned for this class of images (it is explicitly not tuned, for example, for *halftoned* bi-level images). The *1D* Group 3 algorithm runlength-encodes each scanline and then Huffman-encodes the resulting runlengths. Runlengths in the range 0 to 63 are coded with *terminating* codes. Runlengths in the range 64 to 2623 are coded with *make-up* codes, each representing a multiple of 64, followed by a terminating code. Runlengths exceeding 2623 are coded with multiple make-up codes followed by a terminating code. The Huffman tables are fixed, but are separately tuned for black and white runs (except for make-up codes above 1728, which are common). When possible, the *2D* Group 3 algorithm encodes a scanline as a set of short edge deltas (0, ±1, ±2, ±3) with reference to the previous scanline. The delta symbols are entropy-encoded (so that the zero delta symbol is only one bit long etc.) Edges within a 2D-encoded line which can't be delta-encoded are runlength-encoded, and are identified by a prefix. 1D- and 2D-encoded lines are marked differently. 1D-encoded lines are generated at regular intervals, whether actually required or not, to ensure that the decoder can recover from line noise with minimal image degradation. 2D Group 3 achieves compression ratios of up to 6:1 [28].

The Group 4 Facsimile algorithm [18] losslessly compresses bi-level data for transmission over *error-free* communications lines (i.e. the lines are truly error-free, or error-correction is done at a lower protocol level). The Group 4 algorithm is based on the 2D Group 3 algorithm, with the essential modification that since transmission is assumed to be error-free, 1D-encoded lines are no longer generated at regular intervals as an aid to error-recovery. Group 4 achieves compression ratios ranging from 20:1 to 60:1 for the CCITT set of test images [28].

The design goals and performance of the Group 4 compression algorithm qualify it as a compression algorithm for the bi-level layers. However, its Huffman tables are tuned to a lower scanning resolution (100-400 dpi), and it encodes runlengths exceeding 2623 awkwardly.

8.1.2.4 Contone data compression

The contone layer (CMYK) is either a non-compressed bytestream or is compressed to an interleaved JPEG bytestream. The JPEG bytestream is complete and self-contained. It contains all data required for decompression, including quantization and Huffman tables.

The contone data is optionally converted to YCrCb before being compressed (there is no specific advantage in color-space converting if not compressing). Additionally, the CMY contone pixels are optionally converted (on an individual basis) to RGB before color conversion using R=255-C, G=255-M, B=255-Y. Optional bitwise inversion of the K plane may also be performed. Note that this CMY to RGB conversion is not intended to be accurate for display purposes, but rather for the purposes of later converting to YCrCb. The inverse transform will be applied before printing.

8.1.2.4.1 JPEG compression

The JPEG compression algorithm [23] lossily compresses a contone image at a specified quality level. It introduces imperceptible image degradation at compression ratios below 5:1, and negligible image degradation at compression ratios below 10:1 [29].

JPEG typically first transforms the image into a color space which separates luminance and chrominance into separate color channels. This allows the chrominance channels to be subsampled without appreciable loss because of the human visual system's relatively greater sensitivity to luminance than chrominance. After this first step, each color channel is compressed separately.

The image is divided into 8×8 pixel blocks. Each block is then transformed into the frequency domain via a discrete cosine transform (DCT). This transformation has the effect of concentrating image energy in relatively lower-frequency coefficients, which allows higher-frequency coefficients to be more crudely quantized. This quantization is the principal source of compression in JPEG. Further compression is achieved by ordering coefficients by frequency to maximize the likelihood of adjacent zero coefficients, and then runlength-encoding runs of zeroes. Finally, the runlengths and non-zero frequency coefficients are entropy coded. Decompression is the inverse process of compression.

8.1.2.4.2 Non-compressed format

If the contone data is non-compressed, it must be in a block-based format bytestream with the same pixel order as would be produced by a JPEG decoder. The bytestream therefore consists of a series of 8×8 block of the original image, starting with the top left 8×8 block, and working horizontally across the page (as it will be printed) until the top rightmost 8×8 block, then the next row of 8×8 blocks (left to right) and so on until the lower row of 8×8 blocks (left to right). Each 8×8 block consists of 64 8-bit pixels for color plane 0 (representing 8 rows of 8 pixels in the order top left to bottom right) followed by 64 8-bit pixels for color plane 1 and so on for up to a maximum of 4 color planes.

If the original image is not a multiple of 8 pixels in X or Y, padding must be present (the extra pixel data will be ignored by the setting of margins).

8.1.2.4.3 Compressed format

If the contone data is compressed the first memory band contains JPEG headers (including tables) plus MCUs (minimum coded units). The ratio of space between the various color planes in the JPEG stream is 1:1:1:1. No subsampling is permitted. Banding can be completely arbitrary i.e there can be multiple JPEG images per band or 1 JPEG image divided over multiple bands. The break between bands is only memory alignment based.

8.1.2.4.4 Conversion of RGB to YCrCb (in RIP)

YCrCb is defined as per CCIR 601-1 [20] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding and take account of the actual hardware implementation of the inverse transform within SoPEC.

The exact color conversion computation is as follows:
- $Y^* = (9805/32768)R + (19235/32768)G + (3728/32768)B$
- $Cr^* = (16375/32768)R - (13716/32768)G - (2659/32768)B + 128$
- $Cb^* = -(5529/32768)R - (10846/32768)G + (16375/32768)B + 128$

Y, Cr and Cb are obtained by rounding to the nearest integer. There is no need for saturation since ranges of Y^*, Cr^* and Cb^* after rounding are [0-255], [1-255] and [1-255] respectively. *Note that full accuracy is possible with 24 bits*. See [14] for more information.

S3

SoPEC ASIC

9   Overview

The Small Office Home Office Print Engine Controller (SoPEC) is a page rendering engine ASIC that takes compressed page images as input, and produces decompressed page images at up to 6 channels of bi-level dot data as output. The bi-level dot data is generated for the Memjet bi-lithic printhead. The dot generation process takes account of printhead construction, dead nozzles, and allows for fixative generation.

A single SoPEC can control 2 bi-lithic printheads and up to 6 color channels at 10,000 lines/sec[1], equating to 30 pages per minute. A single SoPEC can perform full-bleed printing of A3, A4 and Letter pages. The 6 channels of colored ink are the expected maximum in a consumer SOHO, or office Bi-lithic printing environment:
- CMY, for regular color printing.
- K, for black text, line graphics and gray-scale printing.
- IR (infrared), for Netpage-enabled [5] applications.
- F (fixative), to enable printing at high speed. Because the bi-lithic printer is capable of printing so fast, a fixative may be required to enable the ink to dry before the page touches the page already printed. Otherwise the pages may bleed on each other. In low speed printing environments the fixative may not be required.

SoPEC is *color space agnostic*. Although it can accept contone data as CMYX or RGBX, where X is an optional 4th channel, it also can accept contone data in any print color space. Additionally, SoPEC provides a mechanism for arbitrary mapping of input channels to output channels, including combining dots for ink optimization, generation of channels based on any number of other channels etc. However, inputs are typically CMYK for contone input, K for the bi-level input, and the optional Netpage tag dots are typically rendered to an infra-red layer. A fixative channel is typically generated for fast printing applications.

SoPEC is *resolution agnostic*. It merely provides a mapping between input resolutions and output resolutions by means of scale factors. The expected output resolution is 1600 dpi, but SoPEC actually has no knowledge of the physical resolution of the Bi-lithic printhead.

SoPEC is *page-length agnostic*. Successive pages are typically split into bands and downloaded into the page store as each band of information is consumed and becomes free.

SoPEC provides an interface for synchronization with other SoPECs. This allows simple multi-SoPEC solutions for simultaneous A3/A4/Letter duplex printing. However, SoPEC is also capable of printing only a portion of a page image. Combining synchronization functionality with partial page rendering allows multiple SoPECs to be readily combined for alternative printing requirements including simultaneous duplex printing and wide format printing.

Table 8 lists some of the features and corresponding benefits of SoPEC.

Table 8. Features and Benefits of SoPEC

Features	Benefits
Optimised print architecture in hardware	30ppm full page photographic quality color printing from a desktop PC
0.13micron CMOS (>3 million transistors)	High speed Low cost High functionality

---

1. 10,000 lines per second equates to 30 A4/Letter pages per minute at 1600 dpi

Table 8. Features and Benefits of SoPEC

Feature	Benefits
900 Million dots per second	Extremely fast page generation
10,000 lines per second at 1600 dpi	0.5 A4/Letter pages per SoPEC chip per second
1 chip drives up to 133,920 nozzles	Low cost page-width printers
1 chip drives up to 6 color planes	99% of SoHo printers can use 1 SoPEC device
Integrated DRAM	No external memory required, leading to low cost systems
Power saving sleep mode	SoPEC can enter a power saving sleep mode to reduce power dissipation between print jobs
JPEG expansion	Low bandwidth from PC Low memory requirements in printer
Lossless bitplane expansion	High resolution text and line art with low bandwidth from PC (e.g. over USB)
Netpage tag expansion	Generates interactive paper
Stochastic dispersed dot dither	Optically smooth image quality No moire effects
Hardware compositor for 6 image planes	Pages composited in real-time
Dead nozzle compensation	Extends printhead life and yield Reduces printhead cost
Color space agnostic	Compatible with all inksets and image sources including RGB, CMYK, spot, CIE L*a*b*, hexachrome, YCrCbK, sRGB and other
Color space conversion	Higher quality / lower bandwidth
Computer interface	USB1.1 interface to Host and ISI interface to ISI-Bridge chip thereby allowing connection to IEEE 1394, Bluetooth etc.
Cascadable in resolution	Printers of any resolution
Cascadable in color depth	Special color sets e.g. hexachrome can be used
Cascadable in image size	Printers of any width up to 16 inches
Cascadable in pages	Printers can print both sides simultaneously
Cascadable in speed	Higher speeds are possible by having each SoPEC print one vertical strip of the page.
Fixative channel data generation	Extremely fast ink drying without wastage
Built-in security	Revenue models are protected
Undercolor removal on dot-by-dot basis	Reduced ink usage
Does not require fonts for high speed operation	No font substitution or missing fonts
Flexible printhead configuration	Many configurations of printheads are supported by one chip type
Drives Bi-lithic printheads directly	No print driver chips required, results in lower cost
Determines dot accurate ink usage	Removes need for physical ink monitoring system in ink cartridges

9.1 PRINTING RATES

The required printing rate for SoPEC is 30 sheets per minute with an inter-sheet spacing of 4 cm. To achieve a 30 sheets per minute print rate, this requires:

300mm x 63.(dot/mm) / 2 sec = 105.8 µseconds per line, with no inter-sheet gap.

340mm x 63 (dot/mm) / 2 sec = 93.3 µseconds per line, with a 4 cm inter-sheet gap.

A printline for an A4 page consists of 13824 nozzles across the page [2]. At a system clock rate of 160 MHz 13824 dots of data can be generated in 86.4 µseconds. Therefore data can be generated fast enough to meet the printing speed requirement. It is necessary to deliver this print data to the print-heads.

Printheads can be made up of 5:5, 6:4, 7:3 and 8:2 inch printhead combinations [2]. Print data is transferred to both print heads in a pair simultaneously. This means the longest time to print a line is determined by the time to transfer print data to the longest print segment. There are 9744 nozzles across a 7 inch printhead. The print data is transferred to the printhead at a rate of 106 MHz (2/3 of the system clock rate) per color plane. This means that it will take 91.9 µs to transfer a single line for a 7:3 printhead configuration. So we can meet the requirement of 30 sheets per minute printing with a 4 cm gap with a 7:3 printhead combination. There are 11160 across an 8 inch printhead. To transfer the data to the printhead at 106 MHz will take 105.3 µs. So an 8:2 printhead combination printing with an inter-sheet gap will print slower than 30 sheets per minute.

9.2   SoPEC BASIC ARCHITECTURE

From the highest point of view the SoPEC device consists of 3 distinct subsystems
- CPU Subsystem
- DRAM Subsystem
- Print Engine Pipeline (PEP) Subsystem

See Figure 13 for a block level diagram of SoPEC.

9.2.1   CPU Subsystem

The CPU subsystem controls and configures all aspects of the other subsystems. It provides general support for interfacing and synchronising the external printer with the internal print engine. It also controls the low speed communication to the QA chips. The CPU subsystem contains various peripherals to aid the CPU, such as GPIO (includes motor control), interrupt controller, LSS Master and general timers. The Serial Communications Block (SCB) on the CPU subsystem provides a full speed USB1.1 interface to the Host as well as an Inter SoPEC Interface (ISI) to other SoPEC devices.

9.2.2   DRAM Subsystem

The DRAM subsystem accepts requests from the CPU, Serial Communications Block (SCB) and blocks within the PEP subsystem. The DRAM subsystem (in particular the DIU) arbitrates the various requests and determines which request should win access to the DRAM. The DIU arbitrates based on configured parameters, to allow sufficient access to DRAM for all requestors. The DIU also hides the implementation specifics of the DRAM such as page size, number of banks, refresh rates etc.

9.2.3   Print Engine Pipeline (PEP) subsystem

The Print Engine Pipeline (PEP) subsystem accepts compressed pages from DRAM and renders them to bi-level dots for a given print line destined for a printhead interface that communicates directly with up to 2 segments of a bi-lithic printhead.

The first stage of the page expansion pipeline is the CDU, LBD and TE. The CDU expands the JPEG-compressed contone (typically CMYK) layer, the LBD expands the compressed bi-level layer (typically K), and the TE encodes Netpage tags for later rendering (typically in IR or K ink). The output from the first stage is a set of buffers: the CFU, SFU, and TFU. The CFU and SFU buffers are implemented in DRAM.

The second stage is the HCU, which dithers the contone layer, and composites position tags and the bi-level spot0 layer over the resulting bi-level dithered layer. A number of options exist for the way in which compositing occurs. Up to 6 channels of bi-level data are produced from this stage. Note that not all 6 channels may be present on the printhead. For example, the printhead may be CMY only, with K pushed into the CMY channels and IR ignored. Alternatively, the position tags may be printed in K if IR ink is not available (or for testing purposes).

The third stage (DNC) compensates for dead nozzles in the printhead by color redundancy and error diffusing dead nozzle data into surrounding dots.

The resultant bi-level 6 channel dot-data (typically CMYK-IRF) is buffered and written out to a set of line buffers stored in DRAM via the DWU.

Finally, the dot-data is loaded back from DRAM, and passed to the printhead interface via a dot FIFO. The dot FIFO accepts data from the LLU at the system clock rate (*pclk*), while the PHI removes data from the FIFO and sends it to the printhead at a rate of 2/3 times the system clock rate (see Section 9.1).
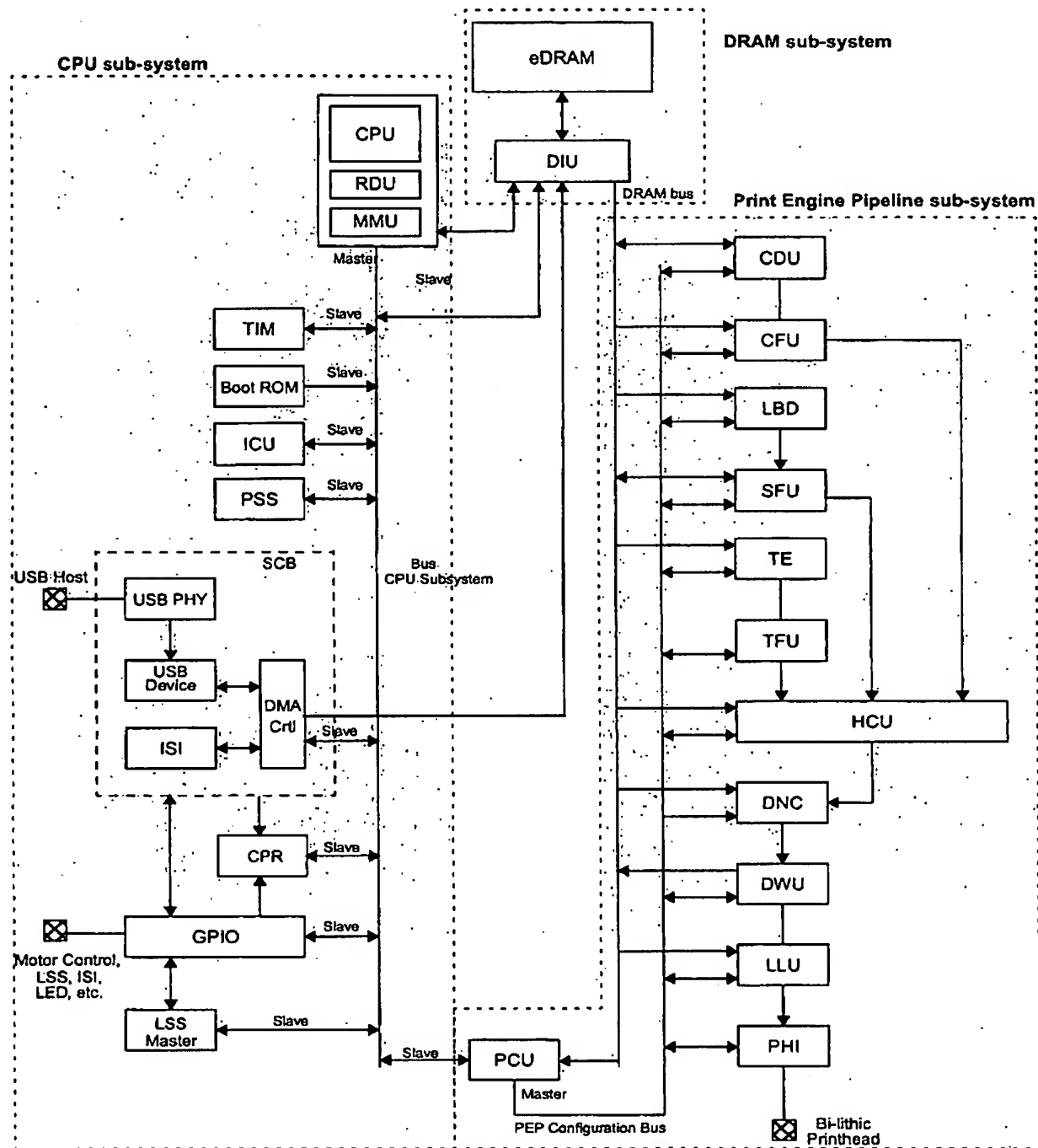
Figure 13. SoPEC System Top Level partition

9.3   SOPEC BLOCK DESCRIPTION

Looking at Figure 13, the various units are described here in summary form:

Table 9. Units within SoPEC

Subsystem	Unit Acronym	Unit Name	Description
DRAM	DIU	DRAM interface unit	Provides the interface for DRAM read and write access for the various SoPEC units, CPU and the SCB block. The DIU provides arbitration between competing units controls DRAM access.
	DRAM	Embedded DRAM	20Mbits of embedded DRAM.
CPU	CPU	Central Processing Unit	CPU for system configuration and control
	MMU	Memory Management Unit	Limits access to certain memory address areas in CPU user mode
	RDU	Real-time Debug Unit	Facilitates the observation of the contents of most of the CPU addressable registers in SoPEC in addition to some pseudo-registers in realtime.
	TIM	General Timer	Contains watchdog and general system timers
	LSS	Low Speed Serial Interfaces	Low level controller for interfacing with the QA chips
	GPIO	General Purpose IOs	General IO controller, with built-in Motor control unit, LED pulse units and de-glitch circuitry
	ROM	Boot ROM	16 KBytes of System Boot ROM code
	ICU	Interrupt Controller Unit	General Purpose interrupt controller with configurable priority, and masking.
	CPR	Clock, Power and Reset block	Central Unit for controlling and generating the system clocks and resets and powerdown mechanisms
	PSS	Power Save Storage	Storage retained while system is powered down
	USB	Universal Serial Bus Device	USB device controller for interfacing with the Host USB.
	ISI	Inter-SoPEC Interface	ISI controller for data and control communication with other SoPEC's in a multi-SoPEC system
	SCB	Serial Communication Block	Contains both the USB and ISI blocks.

Table 9. Units within SoPEC

Subsystem	Unit Acronym	Unit Name	Description
Print Engine Pipeline (PEP)	PCU	PEP controller	Provides external CPU with the means to read and write PEP Unit registers, and read and write DRAM in single 32-bit chunks.
	CDU	Contone decoder unit	Expands JPEG compressed contone layer and writes decompressed contone to DRAM
	CFU	Contone FIFO Unit	Provides line buffering between CDU and HCU
	LBD	Lossless Bi-level Decoder	Expands compressed bi-level layer.
	SFU	Spot FIFO Unit	Provides line buffering between LBD and HCU
	TE	Tag encoder	Encodes tag data into line of tag dots.
	TFU	Tag FIFO Unit	Provides tag data storage between TE and HCU
	HCU	Halftoner compositor unit	Dithers contone layer and composites the bi-level spot 0 and position tag dots.
	DNC	Dead Nozzle Compensator	Compensates for dead nozzles by color redundancy and error diffusing dead nozzle data into surrounding dots.
	DWU	Dotline Writer Unit	Writes out the 6 channels of dot data for a given printline to the line store DRAM
	LLU	Line Loader Unit	Reads the expanded page image from line store, formatting the data appropriately for the bi-lithic printhead.
	PHI	PrintHead Interface	Is responsible for sending dot data to the bi-lithic printheads and for providing line synchronization between multiple SoPECs. Also provides test interface to printhead such as temperature monitoring and Dead Nozzle Identification.

9.4   ADDRESSING SCHEME IN SoPEC

SoPEC must address
- 20 Mbit DRAM.
- PCU addressed registers in PEP.
- CPU-subsystem addressed registers.

SoPEC has a unified address space with the CPU capable of addressing all CPU-subsystem and PCU-bus accessible registers (in PEP) and all locations in DRAM. The CPU generates byte-aligned addresses for the whole of SoPEC.

22 bits are sufficient to byte address the whole SoPEC address space.

9.4.1   DRAM addressing scheme

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbits of DRAM.

Most blocks read or write 256-bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are
- CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.
- The CPU-subsystem always generates a 22-bit byte-aligned DIU address but it will send flags to the DIU indicating whether it is an 8, 16 or 32-bit write.

All DIU accesses must be within the same 256-bit aligned DRAM word.

9.4.2   PEP Unit DRAM addressing

PEP Unit configuration registers which specify DRAM locations should specify 256-bit aligned DRAM addresses i.e. using address bits 21:5. Legacy blocks from PEC1 e.g. the LBD and TE may need to specify 64-bit aligned DRAM addresses if these reused blocks DRAM addressing is difficult to modify. These 64-bit aligned addresses require address bits 21:3. However, these 64-bit aligned addresses should be programmed to start at a 256-bit DRAM word boundary.

Unlike PEC1, there are no constraints in SoPEC on data organization in DRAM except that all data structures must start on a 256-bit DRAM boundary. If data stored is not a multiple of 256-bits then the last word should be padded.

9.4.3   CPU-bus addressed registers

The CPU-bus supports 32-bit word aligned read and write accesses with variable access timings. See section 11.4 for more details of the access protocol used on this bus. The CPU-bus does not currently support byte reads and writes but this can be added at a later date if required by imported IP.

9.4.4   PCU addressed registers in PEP

The PCU only supports 32-bit register reads and writes for the PEP blocks. As the PEP blocks only occupy a subsection of the overall address map and the PCU is explicitly selected by the MMU when a PEP block is being accessed the PCU does not need to perform a decode of the higher-order address bits. See Table 11 for the PEP subsystem address map.

9.5 SoPEC MEMORY MAP

9.5.1 Main memory map

The system wide memory map is shown in Figure 14 below. The memory map is discussed in detail in Section 11 Central Processing Unit (CPU).
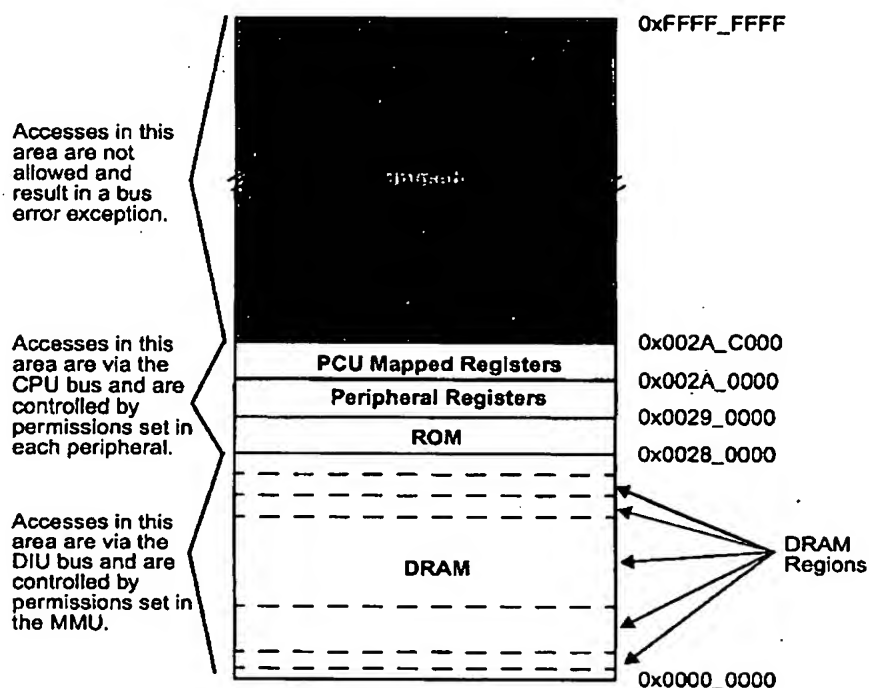
Figure 14. Proposed SoPEC CPU memory map (not to scale)

9.5.2 CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 10 below. The MMU performs the decode of *cpu_adr[21:12]* to generate the relevant *cpu_block_select* signal for each block. The addressed blocks decode however many of the lower order bits of *cpu_adr[11:2]* are required to address all the registers within the block.

Table 10. CPU-bus peripherals address map

Block base	Address
MMU_base	0x0029_0000
TIM_base	0x0029_1000
LSS_base	0x0029_2000
GPIO_base	0x0029_3000
SCB_base	0x0029_4000

Table 10. CPU-bus peripherals address map

Block base	Address
ICU_base	0x0029_5000
CPR_base	0x0029_6000
ROM_base	0x0029_7000
DIU_base	0x0029_8000
PSS_base	0x0029_9000
Reserved	0x0029_A000 to 0x0029_FFFF
PCU_base	0x002A_0000 to 0x002A_BFFF

9.5.3 PCU Mapped Registers (PEP blocks) address map

The PEP blocks are addressed via the PCU. From Figure 14, the PCU mapped registers are in the range 0x002A_0000 to 0x002A_BFFF. From Table 11 it can be seen that there are 12 sub-blocks within the PCU address space. Therefore, only four bits are necessary to address each of the sub-blocks within the PEP part of SoPEC. A further 12 bits may be used to address any configurable register within a PEP block. This gives scope for 1024 configurable registers per sub-block (the PCU mapped registers are all 32-bit addressed registers so the upper 10 bits are required to individually address them). This address will come either from the CPU or from a command stored in DRAM. The bus is assembled as follows:

- address[15:12] = sub-block address,
- address[n:2] = register address within sub-block, only the number of bits required to decode the registers within each sub-block are used,
- address[1:0] = byte address, unused as PCU mapped registers are all 32-bit addressed registers.

So for the case of the HCU, its addresses range from 0x7000 to 0x7FFF within the PEP subsystem or from 0x002A_7000 to 0x002A_7FFFF in the overall system.

Table 11. PEP blocks address map

Block base	Address
PCU_base	0x002A_0000
CDU_base	0x002A_1000
CFU_base	0x002A_2000
LBD_base	0x002A_3000
SFU_base	0x002A_4000
TE_base	0x002A_5000
TFU_base	0x002A_6000
HCU_base	0x002A_7000
DNC_base	0x002A_8000
DWU_base	0x002A_9000
LLU_base	0x002A_A000
PHI_base	0x002A_B000 to 0x002A_BFFF

9.6   BUFFER MANAGEMENT IN SoPEC

As outlined in Section 9.1, SoPEC has a requirement to print 1 side every 2 seconds i.e. 30 sides per minute.

9.6.1   Page buffering

Approximately 2 Mbytes of DRAM are reserved for compressed page buffering in SoPEC. If a page is compressed to fit within 2 Mbyte then a complete page can be transferred to DRAM before printing. However, the time to transfer 2 Mbyte using USB 1.1 is approximately 2 seconds. The worst case cycle time to print a page then approaches 4 seconds. This reduces the worst-case print speed to 15 pages per minute.

9.6.2   Band buffering

The SoPEC page-expansion blocks support the notion of page banding. The page can be divided into bands and another band can be sent down to SoPEC while we are printing the current band.

Therefore we can start printing once at least one band has been downloaded.

The band size granularity should be carefully chosen to allow efficient use of the USB bandwidth and DRAM buffer space. It should be small enough to allow seamless 30 sides per minute printing but not so small as to introduce excessive CPU overhead in orchestrating the data transfer and parsing the band headers. Band-finish interrupts have been provided to notify the CPU of free buffer space. It is likely that the Host PC will supervise the band transfer and buffer management instead of the SoPEC CPU.

If SoPEC starts printing before the complete page has been transferred to memory there is a risk of a buffer underrun occurring if subsequent bands are not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth. A buffer underrun occurs if a line synchronisation pulse is received before a line of data has been transferred to the printhead and causes the print job to fail at that line. If there is no risk of buffer underrun then printing can safely start once at least one band has been downloaded.

If there is a risk of a buffer underrun occurring due to an interruption of compressed page data transfer, then the safest approach is to only start printing once we have loaded up the data for a complete page. This means that a worst case latency in the region of 2 seconds (with USB1.1) will be incurred before printing the first page. Subsequent pages will take 2 seconds to print giving us the required sustained printing rate of 30 sides per minute.

A Storage SoPEC (Section 7.2.5) could be added to the system to provide guaranteed bandwidth data delivery. The print system could also be constructed using an ISI-Bridge chip (Section 7.2.6) to provide guaranteed data delivery.

The most efficient page banding strategy is likely to be determined on a per page/ print job basis and so SoPEC will support the use of bands of any size.

10 SoPEC Use Cases

10.1 INTRODUCTION

This chapter is intended to give an overview of a representative set of scenarios or *use cases* which SoPEC can perform. SoPEC is by no means restricted to the particular use cases described here.

In this chapter we discuss SoPEC use cases under four headings:
1) Normal operation use cases.
2) Security use cases.
3) Miscellaneous use cases.
4) Failure mode use cases.

Use cases for both single and multi-SoPEC systems are outlined.

Some tasks may be composed of a number of sub-tasks.

The realtime requirements for SoPEC software tasks are discussed in "Central Processing Unit (CPU)" under Section 11.3 Realtime requirements.

10.2 NORMAL OPERATION IN A SINGLE SoPEC SYSTEM WITH USB HOST CONNECTION

SoPEC operation is broken up into a number of sections which are outlined below. Buffer management in a SoPEC system is normally performed by the Host.

10.2.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:
1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.
3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation. USB Wakeup.
4) Download and authentication of program (see Section 10.5.2).
5) Store reusable authentication results in Power-Safe Storage (PSS).
6) Execution of program from DRAM.
7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
8) Download and authenticate any further *datasets*.

10.2.2 USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block (chapter 16). Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In a single SoPEC system, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:
1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.

4) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.2).

5) Execution of program from DRAM.

6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.

7) Download and authenticate using results in PSS of any further *datasets* (programs).

10.2.3   Print Initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:

1) Check amount of ink remaining via QA chips.

2) Download static data e.g. dither matrices, dead nozzle tables from Host to DRAM.

3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.

4) Initiate printhead pre-heat sequence, if required.

10.2.4   First page download

Buffer management in a SoPEC system is normally performed by the Host.

First page, first band download and processing:

1) The Host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.

2) The Host downloads the first band (with the page header) to DRAM.

3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and writes directly to PEP registers or to DRAM.

4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.

2) Download the next band with the band header to DRAM.

3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.2.5   Start printing

1) Wait until at least one band of the first page has been downloaded.
One approach is to only start printing once we have loaded up the data for a complete page. If we start printing before the complete page has been transferred to memory we run the risk of a buffer underrun occurring because compressed page data was not transferred to SoPEC in time e.g. due to insufficient USB bandwidth caused by another USB peripheral consuming USB bandwidth.

2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes. A rapid startup order for the PEP units is outlined in Table 12.

Table 12. Typical PEP Unit startup order for printing a page.

Step	Unit
1	DNC
2	DWU
3	HCU

Table 12. Typical PEP Unit startup order for printing a page.

Step	Unit
4	PHI
5	LLU
6	CFU, SFU, TFU
7	CDU
8	TE, LBD

3) Print ready interrupt occurs (from PHI).
4) Start motor control, if first page, otherwise feed the next page. This step could occur before the print ready interrupt.
5) Drive LEDs, monitor paper status.
6) Wait for page alignment via page sensor(s) GPIO interrupt.
7) CPU instructs PHI to start producing line syncs and hence commence printing, or wait for an external device to produce line syncs.
8) Continue to download bands and process page and band headers for next page.

10.2.6 Next page(s) download

As for first page download, performed during printing of current page.

10.2.7 Between bands

When the finished band flags are asserted band related registers in the CDU, LBD, TE need to be re-programmed before the subsequent band can be printed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or most likely by updating from shadow registers. The finished band flag interrupts the CPU to tell the CPU that the area of memory associated with the band is now free.

10.2.8 During page print

Typically during page printing ink usage is communicated to the QA chips.
1) Calculate ink printed (from PHI).
2) Decrement ink remaining (via QA chips).
3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.2.9 Page finish

These operations are typically performed when the page is finished:
1) Page finished interrupt occurs from PHI.
2) Shutdown the PEP blocks by de-asserting their Go registers. A typical shutdown order is defined in Table 13. This will set the PEP Unit state-machines to their idle states without resetting their configuration registers.
3) Communicate ink usage to QA chips, if required.

Table 13. End of page shutdown order for PEP Units (TBD).

Step	Unit
1	PHI (will shutdown by itself in the normal case at the end of a page)
2	DWU (shutting this down stalls the DNC and therefore the HCU and above)
3	LLU (should already be halted due to PHI at end of last line of page)
4	TE (this is the only dot supplier likely to be running, halted by the HCU)
5	CDU (this is likely to already be halted due to end of contone band)
6	CFU, SFU, TFU, LBD (order unimportant, and should already be halted due to end of band)
7	HCU, DNC (order unimportant, should already have halted)

10.2.10 Start of next page

These operations are typically performed before printing the next page:
1) Re-program the PEP Units via PCU command processing from DRAM based on page header.
2) Go to Start printing.

10.2.11 End of document

1) Stop motor control.

10.2.12 Powerdown

In this mode SoPEC is no longer powered.
1) Instruct Host PC via USB that SoPEC is about to power down.

10.2.13 Sleep

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block described in Section 16.
1) Instruct Host PC via USB that SoPEC is about to sleep.
2) Put SoPEC into defined sleep mode.

10.3 NORMAL OPERATION IN A MULTI-SOPEC SYSTEM - ISIMASTER SOPEC

In a multi-SoPEC system the Host generally manages program and compressed page download to all the SoPECs. Inter-SoPEC communication is over the ISI link which will add a latency.

In the case of a multi-SoPEC system with a USB 1.1 connection, the SoPEC with the USB connection is the ISIMaster. The ISI-bridge chip is the ISIMaster in the case of an ISI-Bridge SoPEC configuration.

In a multi-SoPEC system one of the SoPECs will be the PrintMaster. This SoPEC must manage and control sensors and actuators e.g. motor control. These sensors and actuators could be distributed over all the SoPECs in the system. An ISIMaster SoPEC may also be the PrintMaster SoPEC.

In a multi-SoPEC system each printing SoPEC will generally have its own PRINTER_QA chip (or at least access to a PRINTER_QA chip that contains the SoPEC's SOPEC_id_key) to validate operating parameters and ink usage. The results of these operations may be communicated to the PrintMaster SoPEC.

In general the ISIMaster may need to be able to:
- Send messages to the ISISlaves which will cause the ISISlaves to send their status to the ISIMaster.
- Instruct the ISISlaves to perform certain operations.

As the ISI is an insecure interface commands issued over the ISI are regarded as *user mode* commands. *Supervisor mode* code running on the SoPEC CPUs will allow or disallow these commands. The software protocol needs to be constructed with this in mind.

Existing requirements indicate that it is sufficient for the ISIMaster to initiate all communication with the ISISlaves.

SoPEC operation is broken up into a number of sections which are outlined below.

10.3.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.
1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.
3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation USB Wakeup
4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.
5) Download and authentication of program (see Section 10.5.3).
6) Store reusable cryptographic results in Power-Safe Storage (PSS).
7) Execution of program from DRAM.
8) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
9) Download and authenticate any further *datasets* (programs).
10) The initial *dataset* may be broadcast to all the ISISlaves.
11) ISIMaster master SoPEC then waits for a short time to allow the authentication to take place on the ISISlave SoPECs.
12) Each ISISlave SoPEC is polled for the result of its program code authentication process.
13) If all ISISlaves report successful authentication the OEM code module can be distributed and authenticated. OEM could will most likely reside on one SoPEC.

10.3.2 USB wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. For an ISIMaster SoPEC, wakeup can be initiated following a USB reset from the SCB.

A typical USB wakeup sequence is:
1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.
4) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.
5) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).
6) Execution of program from DRAM.
7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
8) Download and authenticate any further *datasets* (programs) using results in Power-Safe Storage (PSS) (see Section 10.5.3).
9) Following steps as per Powerup.

10.3.3   Print initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:
1) Check amount of ink remaining via QA chips which may be present on a ISISlave SoPEC.
2) Download static data e.g. dither matrices, dead nozzle tables from Host to DRAM.
3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly. Instruct ISISlaves to also perform this operation.
4) Initiate printhead pre-heat sequence, if required. Instruct ISISlaves to also perform this operation

10.3.4   First page download

Buffer management in a SoPEC system is normally performed by the Host.
1) The Host communicates to the SoPEC CPU over the USB to check that DRAM space remaining is sufficient to download the first band.
2) The Host downloads the first band (with the page header) to DRAM.
3) When the complete page header has been downloaded the SoPEC CPU processes the page header, calculates PEP register commands and write directly to PEP registers or to DRAM.
4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

Remaining first page bands download and processing:
1) Check DRAM space remaining is sufficient to download the next band.
2) Download the next band with the band header to DRAM.
3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

Poll ISISlaves for DRAM status and download compressed data to ISISlaves.

10.3.5   Start printing

1) Wait until at least one band of the first page has been downloaded.
2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the suggested order defined in Table 12.
3) Print ready interrupt occurs (from PHI). Poll ISISlaves until print ready interrupt.

4) Start motor control (which may be on an ISISlaves SoPEC), if first page, otherwise feed the next page. This step could occur before the print ready interrupt.

5) Drive LEDS, monitor paper status (which may be on an ISISlaves SoPEC).

6) Wait for page alignment via page sensor(s) GPIO interrupt (which may be on an ISISlaves SoPEC).

7) CPU instructs PHI to start producing master line syncs, or wait for an external device to produce line syncs.

8) Continue to download bands and process page and band headers for next page.

10.3.6   Next page(s) download

As for first page download, performed during printing of current page.

10.3.7   Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be re-programmed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or by updating from shadow registers. The finished band flag interrupts to the CPU, tell the CPU that the area of memory associated with the band is now free.

10.3.8   During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).

2) Decrement ink remaining (via QA chips).

3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.3.9   Page finish

These operations are typically performed when the page is finished:

1) Page finished interrupt occurs from PHI. Poll ISISlaves for page finished interrupts.

2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table 13. This will set the PEP Unit state-machines to their startup states.

3) Communicate ink usage to QA chips, if required.

10.3.10   Start of next page

These operations are typically performed before printing the next page:

1) Re-program the PEP Units via PCU command processing from DRAM based on page header.

2) Go to Start printing.

10.3.11   End of document

1) Stop motor control. This may be on an ISISlave SoPEC.

10.3.12   Powerdown

In this mode SoPEC is no longer powered.

1) Instruct Host PC via USB that SoPEC system is about to power down.

2) Instruct ISISlave SoPECs to powerdown.

3) Powerdown ISIMaster SoPEC.

10.3.13   Sleep

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16].

1) Instruct Host PC via USB which parts of SoPEC system are about to sleep.

2) Put defined SoPECs into defined sleep modes.

10.4 NORMAL OPERATION IN A MULTI-SoPEC SYSTEM - ISISLAVE SoPEC

This section the outline typical operation of an ISISlave SoPEC in a multi-SoPEC system. The ISIMaster can be another SoPEC or an ISI-Bridge chip. The ISISlave communicates with the Host via the ISIMaster. Buffer management in a SoPEC system is normally performed by the Host.

10.4.1 Powerup

Powerup describes SoPEC initialisation following an external reset or the watchdog timer system reset.

A typical powerup sequence is:
1) Execute reset sequence for complete SoPEC.
2) CPU boot from ROM.
3) Basic configuration of CPU peripherals, SCB and DIU. DRAM initialisation.
4) Download and authentication of program (see Section 10.5.3).
5) Store reusable cryptographic results in Power-Safe Storage (PSS).
6) Execution of program from DRAM.
7) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
8) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.
9) Download and authenticate any further *datasets*.

10.4.2 ISI wakeup

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16]. Normally the CPU sub-system and the DRAM will be put in sleep mode but the SCB and power-safe storage (PSS) will still be enabled.

Wakeup describes SoPEC recovery from sleep mode with the SCB and power-safe storage (PSS) still enabled. In an ISISlave SoPEC, wakeup can be initiated following an ISI reset from the SCB.

A typical ISI wakeup sequence is:
1) Execute reset sequence for sections of SoPEC in sleep mode.
2) CPU boot from ROM, if CPU-subsystem was in sleep mode.
3) Basic configuration of CPU peripherals and DIU, and DRAM initialisation, if required.
4) Download and authentication of program using results in Power-Safe Storage (PSS) (see Section 10.5.3).
5) Execution of program from DRAM.
6) Retrieve operating parameters from PRINTER_QA and authenticate operating parameters.
7) SoPEC identification by sampling GPIO pins to determine ISIId. Communicate ISIId to ISIMaster.
8) Download and authenticate any further *datasets*.

10.4.3 Print Initialization

This sequence is typically performed at the start of a print job following powerup or wakeup:
1) Check amount of ink remaining via QA chips.
2) Download static data e.g. dither matrices, dead nozzle tables from ISIMaster to DRAM.
3) Check printhead temperature, if required, and configure printhead with firing pulse profile etc. accordingly.
4) Initiate printhead pre-heat sequence, if required.

10.4.4　First page download

Buffer management in a SoPEC system is normally performed by the Host via the ISIMaster.

1) Check DRAM space remaining is sufficient to download the first band.
2) The Host downloads the first band (with the page header) to DRAM via the ISIMaster.
3) When the complete page header has been downloaded, process the page header, calculate PEP register commands and write directly to PEP registers or to DRAM.
4) If PEP register commands have been written to DRAM, execute PEP commands from DRAM via PCU.

Remaining first page bands download and processing:

1) Check DRAM space remaining is sufficient to download the next band.
2) The Host downloads the first band (with the page header) to DRAM via the ISIMaster.
3) When the complete band header has been downloaded, process the band header according to whichever band-related register updating mechanism is being used.

10.4.5　Start printing

1) Wait until at least one band of the first page has been downloaded.
2) Start all the PEP Units by writing to their *Go* registers, via PCU commands executed from DRAM or direct CPU writes, in the order defined in Table 12.
3) Print ready interrupt occurs (from PHI). Communicate to ISIMaster via ISI link.
4) Start motor control, if attached to this ISISlave, when requested by ISIMaster, if first page, otherwise feed next page. This step could occur before the print ready interrupt
5) Drive LEDS, monitor paper status, if on this ISISlave SoPEC, when requested by ISIMaster
6) Wait for page alignment via page sensor(s) GPIO interrupt, if on this ISISlave SoPEC, and send to ISIMaster.
7) Wait for line sync and commence printing.
8) Continue to download bands and process page and band headers for next page.

10.4.6　Next page(s) download

As for first band download, performed during printing of current page.

10.4.7　Between bands

When the finished band flags are asserted band related registers in the CDU, LBD and TE need to be reprogrammed. This can be via PCU commands from DRAM. Typically only 3-5 commands per decompression unit need to be executed. These registers can also be reprogrammed directly by the CPU or by updating from shadow registers. The finished band flag interrupts to the CPU tell the CPU that the area of memory associated with the band is now free.

10.4.8　During page print

Typically during page printing ink usage is communicated to the QA chips.

1) Calculate ink printed (from PHI).
2) Decrement ink remaining (via QA chips).
3) Check amount of ink remaining (via QA chips). This operation may be better performed while the page is being printed rather than at the end of the page.

10.4.9 Page finish

These operations are typically performed when the page is finished:
1) Page finished interrupt occurs from PHI. Communicate page finished interrupt to ISIMaster.
2) Shutdown the PEP blocks by de-asserting their Go registers in the suggested order in Table 13. This will set the PEP Unit state-machines to their startup states.
3) Communicate ink usage to QA chips, if required.

10.4.10 Start of next page

These operations are typically performed before printing the next page:
1) Re-program the PEP Units via PCU command processing from DRAM based on page header.
2) Go to Start printing.

10.4.11 End of document

Stop motor control, if attached to this ISISlave, when requested by ISIMaster.

10.4.12 Powerdown

In this mode SoPEC is no longer powered.
1) Powerdown ISISlave SoPEC when instructed by ISIMaster.

10.4.13 Sleep

The CPU can put different sections of SoPEC into sleep mode by writing to registers in the CPR block [16].
1) Put SoPEC into defined sleep modes.

10.5 SECURITY USE CASES

Please see the 'SoPEC Security Overview' [9] document for a more complete description of SoPEC security issues. The SoPEC boot operation is described in the ROM chapter of the SoPEC hardware design specification, Section 17.2.

10.5.1 Communication with the QA chips

Communication between SoPEC and the QA chips (i.e. INK_QA and PRINTER_QA) will take place on at least a per power cycle and per page basis. Communication with the QA chips has three principal purposes: validating the presence of genuine QA chips (i.e the printer is using approved consumables), validation of the amount of ink remaining in the cartridge and authenticating the operating parameters for the printer. After each page has been printed, SoPEC is expected to communicate the number of dots fired per ink plane to the QA chipset. SoPEC may also initiate decoy communications with the QA chips from time to time.

Process:
- When validating ink consumption SoPEC is expected to principally act as a conduit between the PRINTER_QA and INK_QA chips and to take certain actions (basically enable or disable printing and report status to Host PC) based on the result. The communication channels are insecure but all traffic is signed to guarantee authenticity.

Known Weaknesses
- All communication to the QA chips is over the LSS interfaces using a serial communication protocol. This is open to observation and so the communication protocol could be reverse engineered. In this case both the PRINTER_QA and INK_QA chips could be replaced by impostor devices (e.g. a single FPGA) that successfully emulated the communication protocol. As this would require physical modification of each printer this is considered to be an acceptably low risk. Any messages that are not signed by one of the symmetric keys (such as the SoPEC_id_key) could be reverse engineered. The imposter device must also have access to the appropriate keys to crack the system.
- If the secret keys in the QA chips are exposed or cracked then the system, or parts of it, is compromised.

Assumptions:
- [1] The QA chips are not involved in the authentication of downloaded SoPEC code
- [2] The QA chip in the ink cartridge (INK_QA) does not directly affect the operation of the cartridge in any way i.e. it does not inhibit the flow of ink etc.
- [3] The INK_QA and PRINTER_QA chips are identical in their virgin state. They only become a INK_QA or PRINTER_QA after their FlashROM has been programmed.

10.5.2 Authentication of downloaded code in a single SoPEC system

Process:
1) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.
2) The program is downloaded to the embedded DRAM.
3) The CPU calculates a SHA-1 hash digest of the downloaded program.
4) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.
5) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as

RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

6) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

7) If the hash values do not match then the Host PC is notified of the failure and software may decide to put the SoPEC device into *powerdown* mode.

8) If the hash values match then the CPU starts executing the downloaded program.

9) If, as is very likely, the downloaded program wishes to download subsequent programs (such as OEM code) it is responsible for ensuring the authenticity of everything it downloads. The downloaded program may contain public keys that are used to authenticate subsequent downloads, thus forming a hierarchy of authentication. The SoPEC ROM does not control these authentications - it is solely concerned with verifying that the first program downloaded has come from a trusted source.

10) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

11) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the Host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

Known Weaknesses:

- If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised. A ROM mask change would be required to reprogram the boot0key.

10.5.3   Authentication of downloaded code in a multi-SoPEC system

10.5.3.1 *ISIMaster SoPEC Process:*

1) SoPEC identification by activity on USB end-points 2-4 indicates it is the ISIMaster.

2) The SCB is configured to broadcast the data received from the Host PC.

3) The program is downloaded to the embedded DRAM and broadcasted to all ISISlave SoPECs over the ISI.

4) The CPU calculates a SHA-1 hash digest of the downloaded program.

5) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

6) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

7) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

8) If the hash values do not match then the Host PC is notified of the failure and software may decide to put the SoPEC device into *powerdown* mode.

9) If the hash values match then the CPU starts executing the downloaded program.

10) It is likely that the downloaded program will poll each ISISlave SoPEC for the result of its authentication process and to determine the number of slaves present.

11) If any slave reports a failed authentication then the ISIMaster communicates this to the Host PC and puts itself into *powerdown* mode.

12) If all ISISlaves report successful authentication then the downloaded program is responsible for the downloading, authentication and distribution of subsequent programs within the multi-SoPEC system.

13) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

14) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC determines that all SoPECs are ready to print. The Host PC is informed that the printer is ready to print and the *Start Printing* use case comes into play.

10.5.3.2 ISISlave SoPEC Process:

1) When the CPU comes out of reset the SCB should still be in slave mode, and the SCB is already configured to receive data from the ISIMaster.

2) The program is downloaded to embedded DRAM.

3) The CPU calculates a SHA-1 hash digest of the downloaded program.

4) The *ResetSrc* register in the CPR block is read to determine whether or not a power-on reset occurred.

5) If a power-on reset occurred the signature of the downloaded code (which needs to be in a known location such as the first or last N bytes of the downloaded code) is decrypted using the Silverbrook public boot0key stored in ROM. This decrypted signature is the expected SHA-1 hash of the accompanying program. The encryption algorithm is likely to be a public key algorithm such as RSA. If a power-on reset did not occur then the expected SHA-1 hash is retrieved from the PSS and the compute intensive decryption is not required.

6) The calculated and expected hash values are compared and if they match then the programs authenticity has been verified.

7) If the hash values do not match, then the ISISlave device will await a new program again, eventually timing out and powering down.

8) If the hash values match then the CPU starts executing the downloaded program.

9) It is likely that the downloaded program will communicate the result of its authentication process to the ISIMaster. The downloaded program is responsible for determining the SoPECs ISIId, receiving and authenticating any subsequent programs.

10) At some subsequent point OEM code starts executing. The Silverbrook supervisor code acts as an O/S to the OEM user mode code. The OEM code must access most SoPEC functionality via system calls to the Silverbrook code.

11) The OEM code is expected to perform some simple 'turn on the lights' tasks after which the master SoPEC is informed that this slave is ready to print. The *Start Printing* use case then comes into play.

Known Weaknesses

- If the Silverbrook private boot0key is exposed or cracked then the system is seriously compromised.
- ISI is an open interface i.e. messages sent over the ISI are in the clear. The communication channels are insecure but all traffic is signed to guarantee authenticity. As all communication over the ISI is controlled by Supervisor code on both the ISIMaster and ISISlave then this also provides some protection against software attacks.

10.5.4 Authentication and upgrade of operating parameters for a printer

The SoPEC IC will be used in a range of printers with different capabilities (e.g. A3/A4 printing, printing speed, resolution etc.). It is expected that some printers will also have a software upgrade capability which would allow a user to purchase a license that enables an upgrade in their printer's capabilities (such as print speed). To facilitate this it must be possible to securely store the operating parameters in the PRINTER_QA chip, to securely communicate these parameters to the SoPEC and to securely reprogram the parameters in the event of an upgrade. Note that each printing SoPEC (as opposed to a SoPEC that is only used for the storage of data) will have its own PRINTER_QA chip (or at least access to a PRINTER_QA that contains the SoPEC's SoPEC_id_key). Therefore both ISIMaster and ISISlave SoPECs will need to authenticate operating parameters.

Process:

1) Program code is downloaded and authenticated as described in sections 10.5.2 and 10.5.3 above.

2) The program code has a function to create the SoPEC_id_key from the unique SoPEC_id that was programmed when the SoPEC was manufactured.

3) The SoPEC retrieves the signed operating parameters from its PRINTER_QA chip. The PRINTER_QA chip uses the SoPEC_id_key (which is stored as part of the pairing process executed during printhead assembly manufacture & test) to sign the operating parameters which are appended with a random number to thwart replay attacks.

4) The SoPEC checks the signature of the operating parameters using its SoPEC_id_key. If this signature authentication process is successful then the operating parameters are considered valid and the overall boot process continues. If not the error is reported to the Host PC.

5) Operating parameters may also be set or upgraded using a second key, the *PrintEngineLicense_key*, which is stored on the PRINTER_QA and used to authenticate the change in operating parameters.

Known Weaknesses:

- It may be possible to retrieve the unique SoPEC_id by placing the SoPEC in test mode and scanning it out. It is certainly possible to obtain it by reverse engineering the device. Either way the SoPEC_id (and by extension the SoPEC_id_key) so obtained is valid only for that specific SoPEC and so printers may only be compromised one at a time by parties with the appropriate specialised equipment. Furthermore even if the SoPEC_id is compromised, the other keys in the system, which protect the authentication of consumables and of program code, are unaffected.

10.6   MISCELLANEOUS USE CASES

There are many miscellaneous use cases such as the following examples. Software running on the SoPEC
CPU or Host will decide on what actions to take in these scenarios.

10.6.1   Disconnect / Re-connect of QA chips.
1) Disconnect of a QA chip between documents or if ink runs out mid-document.
2) Re-connect of a QA chip once authenticated e.g. ink cartridge replacement should allow the system
to resume and print the next document

10.6.2   Page arrives before print ready interrupt.
1) Engage clutch to stop paper until print ready interrupt occurs.

10.6.3   Dead-nozzle table upgrade

This sequence is typically performed when dead nozzle information needs to be updated by performing a
printhead dead nozzle test.
1) Run printhead nozzle test sequence
2) Either Host or SoPEC CPU converts dead nozzle information into dead nozzle table.
3) Store dead nozzle table on Host.
4) Write dead nozzle table to SoPEC DRAM.

10.7   FAILURE MODE USE CASES

10.7.1   System errors and security violations

System errors and security violations are reported to the SoPEC CPU and Host. Software running on the SoPEC CPU or Host will then decide what actions to take.

Silverbrook code authentication failure.
   1) Notify Host PC of authentication failure.
   2) Abort print run.

OEM code authentication failure.
   1) Notify Host PC of authentication failure.
   2) Abort print run.

Invalid QA chip(s).
   1) Report to Host PC.
   2) Abort print run.

MMU security violation interrupt.
   1) This is handled by exception handler.
   2) Report to Host PC
   3) Abort print run.

Invalid address interrupt from PCU.
   1) This is handled by exception handler.
   2) Report to Host PC.
   3) Abort print run.

Watchdog timer interrupt.
   1) This is handled by exception handler.
   2) Report to Host PC.
   3) Abort print run.

Host PC does not acknowledge message that SoPEC is about to power down.
   1) Power down anyway.

10.7.2   Printing errors

Printing errors are reported to the SoPEC CPU and Host. Software running on the Host or SoPEC CPU will then decide what actions to take.

Insufficient space available in SoPEC compressed band-store to download a band.
   1) Report to the Host PC.

Insufficient ink to print.
   1) Report to Host PC.

Page not downloaded in time while printing.
   1) Buffer underrun interrupt will occur.
   2) Report to Host PC and abort print run.

JPEG decoder error interrupt.
   1) Report to Host PC.

CPU SUBSYSTEM

11 Central Processing Unit (CPU)

11.1 OVERVIEW

The CPU block consists of the CPU core, MMU, cache and associated logic. The principal tasks for the program running on the CPU to fulfill in the system are:

Communications:
- Control the flow of data from the USB interface to the DRAM and ISI
- Communication with the host via USB or ISI
- Running the USB device driver

PEP Subsystem Control:
- Page and band header processing (may possibly be performed on host PC)
- Configure printing options on a per band, per page, per job or per power cycle basis
- Initiate page printing operation in the PEP subsystem
- Retrieve dead nozzle information from the printhead interface (PHI) and forward to the host PC
- Select the appropriate firing pulse profile from a set of predefined profiles based on the printhead characteristics
- Retrieve printhead temperature via the PHI

Security:
- Authenticate downloaded program code and printer operating parameters
- Authenticate consumables via the PRINTER_QA and INK_QA chips
- Monitor ink usage
- Isolation of OEM code from direct access to the system resources

Other:
- Drive the printer motors using the GPIO pins
- Monitoring the status of the printer (paper jam, tray empty etc.)
- Driving front panel LEDs
- Perform post-boot initialisation of the SoPEC device
- Memory management (likely to be in conjunction with the host PC)
- Miscellaneous housekeeping tasks

To control the Print Engine Pipeline the CPU is required to provide a level of performance at least equivalent to a 16-bit Hitachi H8-3664 microcontroller running at 16 MHz. An as yet undetermined amount of additional CPU performance is needed to perform the other tasks. The extra performance required is dominated by the signature verification task and the SCB (including the USB) management task. An operating system is not required at present. A number of CPU cores have been evaluated and the LEON P1754 is considered to be the most appropriate solution. A diagram of the CPU block is shown in Figure 15 below.
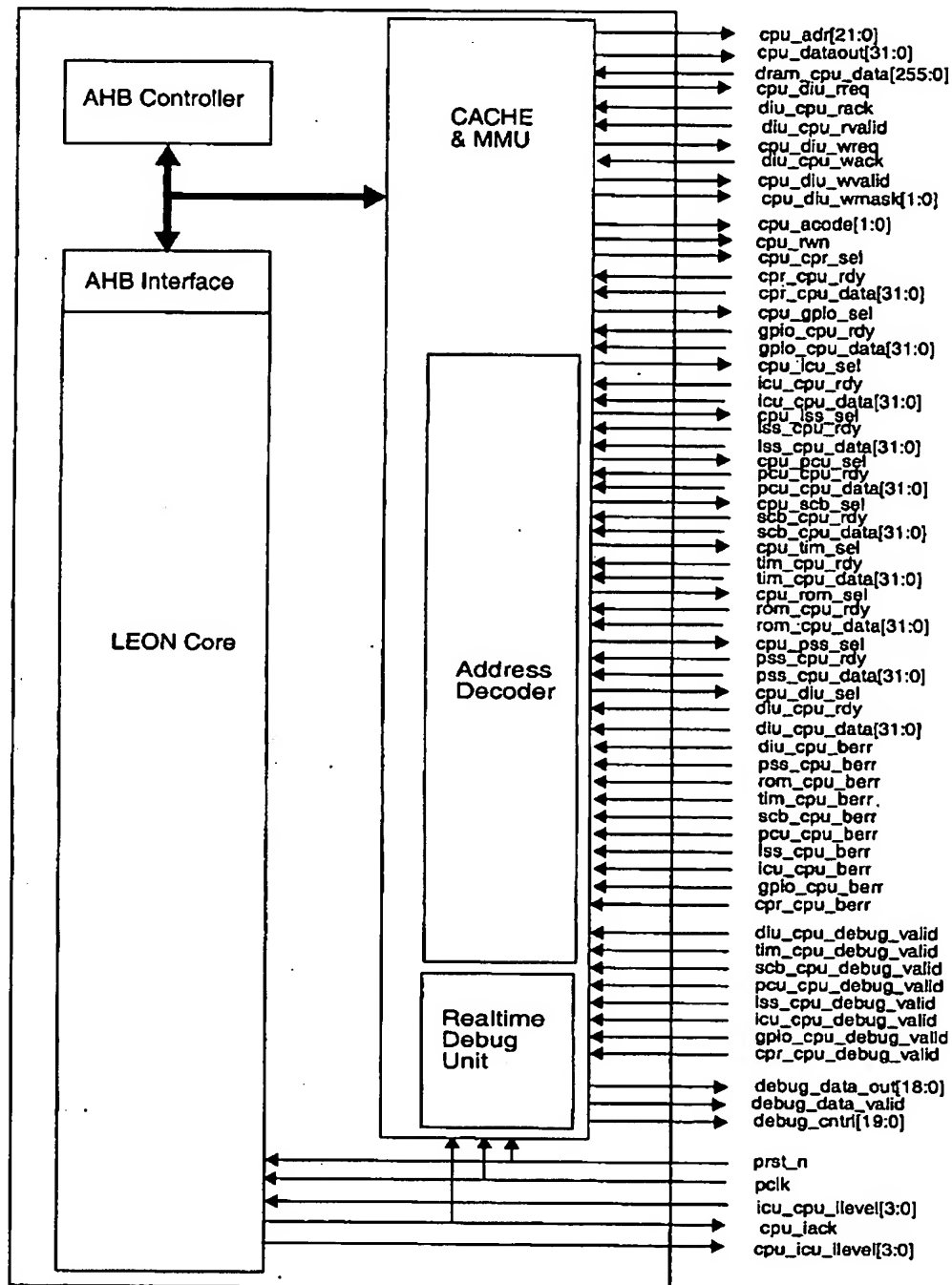
Figure 15. CPU block diagram

11.2   DEFINITIONS OF I/Os

Table 14. CPU Subsystem I/Os

Port name	# pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
pclk	1	In	Global clock
CPU to DIU DRAM Interface			
cpu_adr[21:0]	22	Out	Address bus for both DRAM and peripheral access
cpu_dataout[31:0]	32	Out	Data out to both DRAM and peripheral devices. This should be driven at the same time as the *cpu_adr* and request signals.
dram_cpu_data[255:0]	256	In	Read data from the DRAM
cpu_diu_rreq	1	Out	Read request to the DIU DRAM
diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *dram_cpu_data* bus
cpu_diu_wreq	1	Out	Write request to the DIU
diu_cpu_wack	1	In	Acknowledge from the DIU that the write request has been accepted
cpu_diu_wvalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_dataout* bus is valid
cpu_diu_wmask[1:0]	2	Out	Flag indicating format of CPU write to DRAM *cpu_diu_wmask* = 00: 8-bit write *cpu_diu_wmask* = 01: 16-bit write *cpu_diu_wmask* = 10: 32-bit write *cpu_diu_wmask* = 11: reserved cpu_adr[2:0] are driven in accordance with the width of the data access indicated by *cpu_diu_wmask*. Addresses cannot cross a 256-bit word DRAM boundary.
CPU to peripheral blocks			
cpu_rwn	1	Out	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	Out	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access
cpu_cpr_sel	1	Out	CPR block select.
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	In	CPR bus error signal to the CPU.
cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
cpu_gpio_sel	1	Out	GPIO block select.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
gpio_cpu_berr	1	In	GPIO bus error signal to the CPU.
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
cpu_icu_sel	1	Out	ICU block select.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
icu_cpu_berr	1	In	ICU bus error signal to the CPU.
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block

Table 14. CPU Subsystem I/Os

Port name	Pins	I/O	Description
cpu_lss_sel	1	Out	LSS block select.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
lss_cpu_berr	1	In	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
cpu_pcu_sel	1	Out	PCU block select.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
pcu_cpu_berr	1	In	PCU bus error signal to the CPU.
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
cpu_scb_sel	1	Out	SCB block select.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
scb_cpu_berr	1	In	SCB bus error signal to the CPU.
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
cpu_tim_sel	1	Out	Timers block select.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
tim_cpu_berr	1	In	Timers bus error signal to the CPU.
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
cpu_rom_sel	1	Out	ROM block select.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
rom_cpu_berr	1	In	ROM bus error signal to the CPU.
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
cpu_pss_sel	1	Out	PSS block select.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.
pss_cpu_berr	1	In	PSS bus error signal to the CPU.
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
cpu_diu_sel	1	Out	DIU register block select.
diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
diu_cpu_berr	1	In	DIU bus error signal to the CPU.
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
Interrupt signals			
icu_cpu_ilevel[3:0]	3	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
cpu_icu_ilevel[3:0]	3	Out	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
Debug signals			
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.

Table 14. CPU Subsystem I/Os

Port name	#Pins	I/O	Description
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	18	Out	Output debug data to be muxed on to the PHI pins
debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	20	Out	Control signal for each PHI bound debug data line indicating whether or not the debug data should be selected by the pin mux

11.3 REALTIME REQUIREMENTS

The SoPEC realtime requirements have yet to be fully determined but they may be split into three categories: hard, firm and soft

11.3.1 Hard realtime requirements

Hard requirements are tasks that must be completed before a certain deadline or failure to do so will result in an error perceptible to the user (printing stops or functions incorrectly). There are three hard realtime tasks:

- Motor control: The motors which feed the paper through the printer at a constant speed during printing are driven directly by the SoPEC device. Four periodic signals with different phase relationships need to be generated to ensure the paper travels smoothly through the printer. The generation of these signals is handled by the GPIO hardware (see section 13.2 for more details) but the CPU is responsible for enabling these signals (i.e. to start or stop the motors) and coordinating the movement of the paper with the printing operation of the printhead.

- Buffer management: Data enters the SoPEC via the SCB at an uneven rate and is consumed by the PEP subsystem at a different rate. The CPU is responsible for managing the DRAM buffers to ensure that neither overrun nor underrun occur. This buffer management is likely to be performed under the direction of the host.

- Band processing: In certain cases PEP registers may need to be updated between bands. As the timing requirements are most likely too stringent to be met by direct CPU writes to the PCU a more likely scenario is that a set of shadow registers will programmed in the compressed page units before the current band is finished, copied to band related registers by the finished band signals and the processing of the next band will continue immediately. An alternative solution is that the CPU will construct a DRAM based set of commands (see section 21.8.5 for more details) that can be executed by the PCU. The task for the CPU here is to parse the band headers stored in DRAM and generate a DRAM based set of commands for the next number of bands. The location of the DRAM based set of commands must then be written to the PCU before the current band has been processed by the PEP subsystem. It is also conceivable (but currently considered unlikely) that the host PC could create the DRAM based commands. In this case the CPU will only be required to point the PCU to the correct location in DRAM to execute commands from.

11.3.2 Firm requirements

Firm requirements are tasks that should be completed by a certain time or failure to do so will result in a degradation of performance but not an error. The majority of the CPU tasks for SoPEC fall into this category including all interactions with the QA chips, program authentication, page feeding, configuring PEP registers for a page or job, determining the firing pulse profile, communication of printer status to the host over the USB and the monitoring of ink usage. The authentication of downloaded programs and messages will be the most compute intensive operation the CPU will be required to perform. Initial investigations indicate that the LEON processor, running at 160 MHz, will easily perform three authentications in under a second.

Table 15. Expected firm requirements

Requirement	Duration
Power-on to start of printing first page [USB and slave SoPEC enumeration, 3 or more RSA signature verifications, code and compressed page data download and chip initialisation]	~ 8 secs ??
Wake-up from sleep mode to start printing [3 or more SHA-1 operations, code and compressed page data download and chip re-initialisation	~ 2 secs
Authenticate ink usage in the printer	~ 0.5 secs
Determining firing pulse profile	~ 0.1 secs
Page feeding, gap between pages	OEM dependent
Communication of printer status to host PC	~ 10 ms
Configuring PEP registers	??

11.3.3 Soft requirements

Soft requirements are tasks that need to be done but there are only light time constraints on when they need to be done. These tasks are performed by the CPU when there are no pending higher priority tasks. As the SoPEC CPU is expected to be lightly loaded these tasks will mostly be executed soon after they are scheduled.

11.4 BUS PROTOCOLS

As can be seen from Figure 15 above there are different buses in the CPU block and different protocols are used for each bus. There are three buses in operation:

11.4.1 CPU core to cache/MMU bus

This is the native bus of the CPU core. See section 11.6.6.1 for more details. Timing and full signal details should be provided in the documentation accompanying this core.

11.4.2 Cache/MMU to DIU bus

This bus conforms to the DIU bus protocol described in Section 20.13.2. Note that the address and data buses are shared with the peripheral bus. The effective bus width differs between a read (256 bits) and a write (32/16/8 bits) and only the bottom 32 bits of the bus are shared with the peripheral bus. As certain CPU instructions may require byte write access this will need to be supported in the DIU. See section 11.6.6.2 for more details.

11.4.3   CPU Subsystem Bus

For access to the on-chip peripherals a simple bus protocol is used. The MMU must first determine which particular block is being addressed (and that the access is a valid one) so that the appropriate block select signal can be generated. During a write access CPU write data is driven out with the address and block select signals in the first cycle of an access. The addressed slave peripheral responds by asserting its ready signal indicating that it has registered the write data and the access can complete. The write data bus is common to all peripherals and is also used for CPU writes to the embedded DRAM. A read access is initiated by driving the address and select signals during the first cycle of an access. The addressed slave responds by placing the read data on its bus and asserting its ready signal to indicate to the CPU that the read data is valid. Each block has a separate point-to-point data bus for read accesses to avoid the need for a tri-stateable bus.

All peripheral accesses are 32-bit. Support for byte or 16-bit accesses may be added if required by an imported IP block such as the USB controller. The use of the ready signal allows the accesses to be of variable length. In most cases accesses will complete in two cycles but three or four (or more) cycles accesses are likely for PEP blocks or IP blocks with a different native bus interface. All PEP blocks are accessed via the PCU which acts as a bridge. The PCU bus uses a similar protocol to the CPU subsystem bus but with the PCU as the bus master.

The duration of accesses to the PEP blocks is influenced by whether or not the PCU is executing commands from DRAM. As these commands are essentially register writes the CPU access will need to wait until the PCU bus becomes available when a register access has been completed. This could lead to the CPU being stalled for up to 4 cycles if it attempts to access PEP blocks while the PCU is executing a command. The size and probability of this penalty is sufficiently small to have any significant impact on performance.

In order to support user mode (i.e. OEM code) access to certain peripherals the CPU subsystem bus propagates the CPU function code signals (*cpu_acode[1:0]*). These signals indicate the type of address space (i.e. User/Supervisor and Program/Data) being accessed by the CPU for each access. Each peripheral must determine whether or not the CPU is in the correct mode to be granted access to its registers and in some cases (e.g. Timers and GPIO blocks) different access permissions can apply to different registers within the block. If the CPU is not in the correct mode then the violation is flagged by asserting the block's bus error signal (*block_cpu_berr*) with the same timing as its ready signal (*block_cpu_rdy*) which remains deasserted. When this occurs invalid read accesses should return 0 and write accesses should have no effect.

Figure 16 shows two examples of the peripheral bus protocol in action. A write to the LSS block from code running in supervisor mode is successfully completed. This is immediately followed by a read from a PEP block via the PCU from code running in user mode. As this type of access is not permitted the access is terminated with a bus error. The bus error exception processing then starts directly after this - no further accesses to the peripheral should be required as the exception handler should be located in the DRAM.

Each peripheral acts as a slave on the CPU subsystem bus and its behavior is described by the state machine in section 11.4.3.1
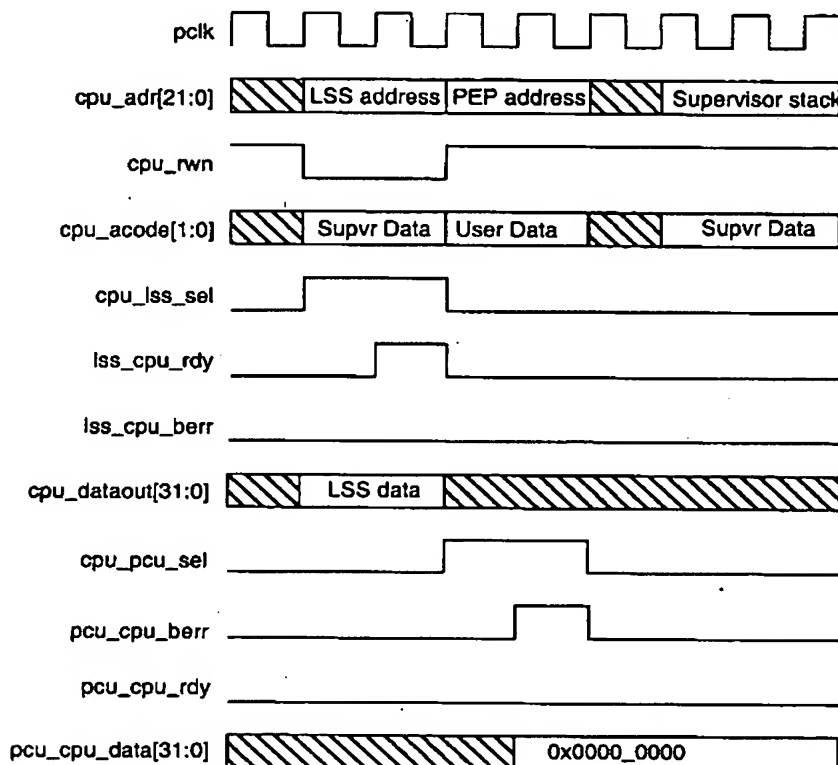
Figure 16. CPU bus transactions

11.4.3.1 CPU subsystem bus slave state machine

CPU subsystem bus slave operation is described by the state machine in Figure 17. This state machine will be implemented in each CPU subsystem bus slave. The only new signals mentioned here are the *valid_access* and *reg_available* signals. The *valid_access* is determined by comparing the *cpu_acode* value with the block or register (in the case of a block that allow user access on a per register basis such as the GPIO block) access permissions and asserting *valid_access* if the permissions agree with the CPU mode. The *reg_available* signal is only required in the PCU or in blocks that are not capable of two-cycle access (e.g. blocks containing imported IP with different bus protocols). In these blocks the *reg_available* signal is an internal signal used to insert wait states (by delaying the assertion of *block_cpu_rdy*) until the CPU bus slave interface can gain access to the register.

When reading from a register that is less than 32 bits wide the CPU susystems bus slave should return zeroes on the unused upper bits of the *block_cpu_data* bus.

To support debug mode the contents of the register selected for debug observation, *debug_reg*, are always output on the *block_cpu_data* bus whenever a read access is not taking place. See section 11.8 for more details of debug operation.
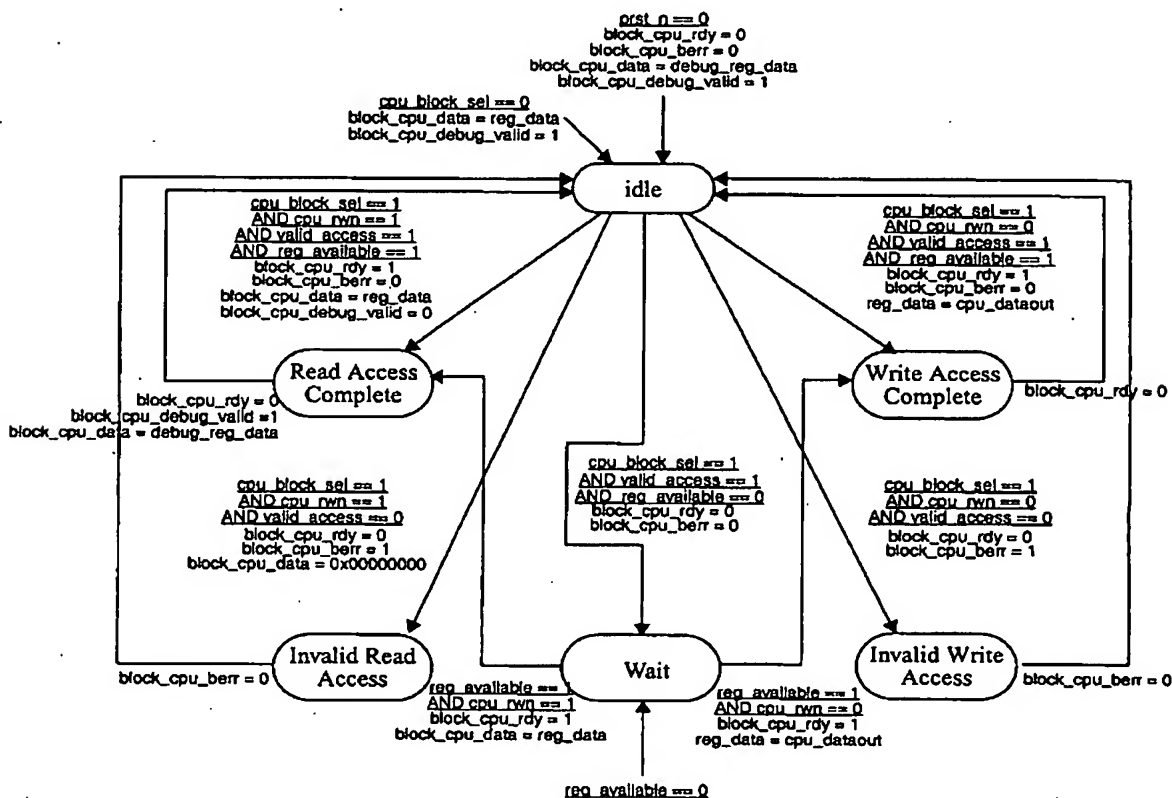
Figure 17. State machine for a CPU subsystem slave

11.5   LEON CPU

The LEON processor is an open-source implementation of the IEEE-1754 standard (SPARC V8) instruction set. LEON is available from and actively supported by Gaisler Research (www.gaisler.com).

The following features of the LEON-2 processor will be utilised on SoPEC:

- IEEE-1754 (SPARC V8) compatible integer unit with 5-stage pipeline
- Separate instruction and data cache (Harvard architecure)
- Set-associative caches: 1-4 sets, 1-64 kbyte/set. Random, LRR or LRU replacement. Direct mapped cacches are also available and are the more likely option for SoPEC.
- Full implementation of AMBA-2.0 AHB on-chip bus
- Power-down mode

The standard release of LEON incorporates a number of peripherals and support blocks which will not be included on SoPEC. The LEON core as used on SoPEC will consist of: 1) the LEON integer unit, 2) possibly the instruction and data caches (currently under review), 3) the cache control logic (to be signifi-

cantly reduced by optimisation if the caches are not used), 4) the AHB interface and 5) possibly the AHB controller (although this functionality may be implemented in the LEON Bridge).

The version of the LEON database that the SoPEC LEON components will be sourced from is LEON2-1.0.8 although later versions may be used if they offer worthwhile functionality or bug fixes that affect the SoPEC design. Note that if the LEON caches are not used then we may revert to v1.0.7 of the database as the cache control logic is likely to be simpler and easier to optimise away (v1.0.8 introduced support for set-associative caching)

The LEON core will be clocked using the system clock, *pclk*, and reset using the *prst_n_section[1]* signal. The ICU will assert all the hardware interrupts using the protocol described in section 11.9. The particular types of SRAMs (for LEON caches) and register files used will be determined during the implementation phase. The LEON hardware multipliers are not expected to be required. Furthermore it is anticipated that SoPEC will use the recommended 8 register window configuration

Further details of the SPARC V8 instruction set and the LEON processor can be found in [32] and [33] respectively.

11.6 MEMORY MANAGEMENT UNIT (MMU)

Memory Management Units are typically used to protect certain regions of memory from invalid accesses, to perform address translation for a virtual memory system and to maintain memory page status (swapped-in, swapped-out or unmapped)

The SoPEC MMU is a much simpler affair whose function is to ensure that all regions of the SoPEC memory map are adequately protected. The MMU does not support virtual memory and physical addresses are used at all times - the one exception to this is the address translation of the reset vector. The SoPEC MMU supports a full 32-bit address space. A proposed memory map is shown in Figure 18 below.

The MMU selects the relevant bus protocol and generate the appropriate control signals depending on the area of memory being accessed. The MMU is responsible for performing the address decode and generation of the appropriate block select signal as well as the selection of the correct block read bus during a read access. The MMU will need to support all of the bus transactions the CPU can produce including interrupt acknowledge cycles, aborted transactions etc.

When an MMU error occurs (such as an attempt to access a supervisor mode only region when in user mode) a bus error is generated. While the LEON can recognise different types of bus error (e.g. data store error, instruction access error) it appears to handle them in the same manner as it handles all traps i.e it will transfer control to a trap handler. No extra state information appears to be stored because of the nature of the trap.The location of the trap handler is contained in the TBR (Trap Base Register). This is the same mechanism as is used to handle interrupts. Further investigation is needed to determine exactly how LEON behaves when a bus error type trap occurs to determine the best approach to handling bus errors. It may be simplest to just treat them as the highest priority interrupt.
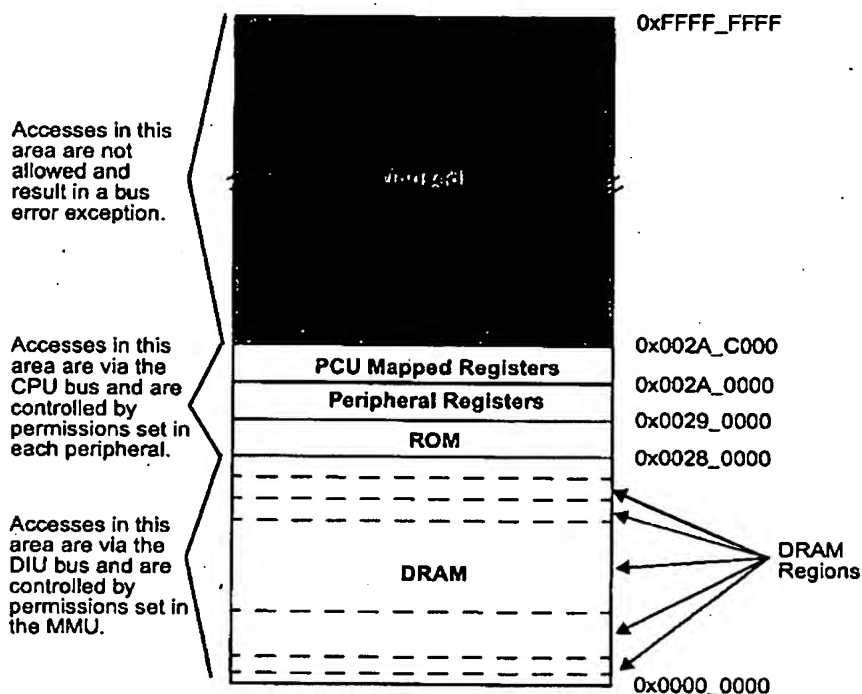
Figure 18. Proposed SoPEC CPU memory map (not to scale)

11.6.1 CPU-bus peripherals address map

The address mapping for the peripherals attached to the CPU-bus is shown in Table 16 below. The MMU performs the decode of the high order bits to generate the relevant *cpu_block_select* signal. Apart from the PCU, which decodes the address space for the PEP blocks, each block only needs to decode as many bits of *cpu_adr[11:2]* as required to address all the registers within the block.

Table 16. CPU-bus peripherals address map

Block base	Address
MMU_base	0x0029_0000
TIM_base	0x0029_1000
LSS_base	0x0029_2000
GPIO_base	0x0029_3000
SCB_base	0x0029_4000
ICU_base	0x0029_5000
CPR_base	0x0029_6000
ROM_base	0x0029_7000
DIU_base	0x0029_8000
PSS_base	0x0029_9000

Table 16. CPU-bus peripherals address map

Block base	Address
Reserved	0x0029_A000 to 0x0029_FFFF
PCU_base	0x002A_0000

11.6.2   DRAM Region Mapping

The embedded DRAM is broken into 8 regions, with each region defined by a lower and upper bound address and with its own access permissions.

The association of an area in the DRAM address space with a MMU region is completely under software control. Table 17 below gives one possible region mapping. Regions should be defined according to their access requirements and position in memory. Regions that share the same access requirements and that are contiguous in memory may be combined into a single region. The example below is purely for indicative purposes - real mappings are likely to differ significantly from this. Note that the RegionBottom and RegionTop fields in this example are byte aligned and would need to be right-shifted by 5 places to obtain the 256-bit aligned value used to program the *RegionNTop* and *RegionNBottom* registers. or more details, see 11.6.5.1 and 11.6.5.2.

Table 17. Example region mapping

Region	RegionBottom	RegionTop	Description
0	0x0000_0000	0x0000_0FFF	Silverbrook OS (supervisor) data
1	0x0000_1000	0x0000_BFFF	Silverbrook OS (supervisor) code
2	0x0000_C000	0x0000_C3FF	Silverbrook (supervisor/user) data
3	0x0000_C400	0x0000_CFFF	Silverbrook (supervisor/user) code
4	0x0026_D000	0x0026_D3FF	OEM (user) data
5	0x0026_D400	0x0026_DFFF	OEM (user) code
6	0x0027_E000	0x0027_FFFF	Shared Silverbrook/OEM space
7	0x0000_0000	0x0026_CFFF	Compressed page store (supervisor data)

11.6.3   Non-DRAM regions

As shown in Figure 18 the DRAM occupies only 2.5 MBytes of the total 4 GB SoPEC address space. The non-DRAM regions of SoPEC are handled by the MMU as follows:

ROM (0x0028_0000 to 0x0028_FFFF): The ROM block will control the access types allowed. The *cpu_acode[1:0]* signals will indicate the CPU mode and access type and the ROM block will assert *rom_cpu_berr* if an attempted access is forbidden. The protocol is described in more detail in section 11.4.3. The ROM block access permissions are hard wired to allow all read accesses except to the *FuseChipID* registers which may only be read in supervisor mode.

MMU Internal Registers (0x0029_0000 to 0x0029_0FFF): The MMU is responsible for controlling the accesses to its own internal registers and will only allow data reads and writes (no instruction fetches) from supervisor data space. All other accesses will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol.

CPU Subsystem Peripheral Registers (0x0029_1000 to 0x0029_FFFF): Each peripheral block will control the access types allowed. Every peripheral will allow supervisor data accesses (both read and write) and some blocks (e.g. Timers and GPIO) will also allow user data space accesses as outlined in the relevant chapters of this specification. Neither supervisor nor user instruction fetch accesses are allowed to

any block as it is not possible to execute code from peripheral registers. The bus protocol is described in section 11.4.3.

PCU Mapped Registers (0x002A_0000 to 0x002A_BFFF): All of the PEP blocks registers which are accessed by the CPU via the PCU will inherit the access permissions of the PCU. These access permissions are hard wired to allow supervisor data accesses only and the protocol used is the same as for the CPU peripherals.

Unused address space (0x002A_C000 to 0xFFFF_FFFF): All accesses to the unused portion of the address space will result in the *mmu_cpu_berr* signal being asserted in accordance with the CPU native bus protocol. These accesses will not propagate outside of the MMU i.e. no external access will be initiated.

11.6.4   Reset exception vector and reference zero traps

When a reset occurs the LEON processor starts executing code from address 0x0000_0000. On SoPEC the embedded DRAM occupies this area of the address map. As the DRAM contents are undefined when the processor comes out of reset (this is certainly the case with a power-on and most other resets that can occur on SoPEC) the MMU will need to redirect accesses from 0x0000_0000 through 0x0000_00?? (the minimum amount of redirection is currently TBD but is likely to be at least 16 bytes) to the bottom of the ROM i.e. to 0x0028_0000 through 0x0028_00??.

A common software bug is zero-referencing or null pointer de-referencing (where the program attempts to access the contents of address 0x0000_0000). To assist software debug the MMU will assert a bus error every time the reset locations are accessed after the reset trap handler has legitimately been retrieved immediately after reset. If desired this condition could be result in a unique trap (e.g. a watchpoint detected trap)

11.6.5   MMU Configuration Registers

These are the only configuration registers in the CPU block. Note that all the MMU configuration registers may only be accessed when the CPU is running in supervisor mode.

Table 18. MMU Configuration Registers

Address offset from MMU base	Register	#bits	Reset	Description
0x00	Region0Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 0
0x04	Region0Top	17	0xF_FFFF	This register contains the physical address that marks the top of region 0. Region 0 covers the entire address space after reset whereas all other regions are zero-sized initially.
0x08	Region1Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 1
0x0C	Region1Top	17	0x0_0000	This register contains the physical address that marks the top of region 1
0x10	Region2Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 2
0x14	Region3Top	17	0x0_0000	This register contains the physical address that marks the top of region 2
0x18	Region3Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 3
0x1C	Region3Top	17	0x0_0000	This register contains the physical address that marks the top of region 3

Table 18. MMU Configuration Registers

Address offset from MMU base	Register	#bits	Reset	Description
0x20	Region4Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 4
0x24	Region4Top	17	0x0_0000	This register contains the physical address that marks the top of region 4
0x28	Region5Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 5
0x2C	Region5Top	17	0x0_0000	This register contains the physical address that marks the top of region 5
0x30	Region6Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 6
0x34	Region6Top	17	0x0_0000	This register contains the physical address that marks the top of region 6
0x38	Region7Bottom	17	0x0_0000	This register contains the physical address that marks the bottom of region 7
0x3C	Region7Top	17	0x0_0000	This register contains the physical address that marks the top of region 7
0x40	Region0Control	6	0x07	Control register for region 0
0x44	Region1Control	6	0x07	Control register for region 1
0x48	Region2Control	6	0x07	Control register for region 2
0x4C	Region3Control	6	0x07	Control register for region 3
0x50	Region4Control	6	0x07	Control register for region 4
0x54	Region5Control	6	0x07	Control register for region 5
0x58	Region6Control	6	0x07	Control register for region 6
0x5C	Region7Control	6	0x07	Control register for region 7
0x60	BusTimeout	16	0x00FF	This register should be set to the number of *pclk* cycles to wait before aborting an access with a bus error.
0x64	DebugSelect	7	0x00	Contains address of the register selected for debug observation. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined during the implementation phase.

11.6.5.1 RegionTop and RegionBottom registers

The 20 Mbit of embedded DRAM on SoPEC is arranged as 81920 words of 256 bits each. All region boundaries need to align with a 256-bit word. Thus only 17 bits are required for the *RegionNTop* and *RegionNBottom* registers. The byte address of these locations can be obtained by simply left-shifting the register value by 5 bits i.e. $cpu\_adr[21:0] = RegionNTop/Bottom[16:0] << 5$.

Both the *RegionNTop* and *RegionNBottom* registers are inclusive i.e. the addresses in the registers are included in the region. The size of smallest active region is therefore 2 256-bit words i.e. 64 bytes.

If DRAM regions overlap (there is no reason for this to be the case but there is nothing to prohibit it either) then only accesses allowed by all overlapping regions are permitted. That is if a DRAM address appears in both Region1 and Region3 (for example) the *cpu_acode* of an access is checked against the access permissions of both regions. If both regions permit the access then it will proceed but if either or both regions do not permit the access then it will not be allowed.

The MMU does not support negatively sized regions i.e. the value of the *RegionNTop* register should always be greater that the value of the *RegionNBottom* register. If *RegionNTop* is lower in the address map than *RegionNTop* then the region is considered to be zero-sized and is ignored.

When both the *RegionNTop* and *RegionNBottom* registers for a region contain the same value the region is then simply one 256-bit word in length and this corresponds to the smallest possible active region.

11.6.5.2 Region Control registers

Each memory region has a control register associated with it. The *RegionNControl* register is used to set the access conditions for the memory region bounded by the *RegionNTop* and *RegionNBottom* registers. Table 19 describes the function of each bit field in the *RegionNControl* registers. All bits in a *RegionNControl* register are both readable and writable by design. However, like all registers in the MMU, the *RegionNControl* registers can only be accessed by code running in supervisor mode.

Table 19. Region Control Register

Field Name	bit(s)	Description
SupervisorAccess	2:0	Denotes the type of access allowed when the CPU is running in Supervisor mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit0 - Data read access permission bit1 - Data write access permission bit2 - Instruction fetch access permission
UserAccess	5:3	Denotes the type of access allowed when the CPU is running in User mode. For each access type a 1 indicates the access is permitted and a 0 indicates the access is not permitted. bit3 - Data read access permission bit4 - Data write access permission bit5 - Instruction fetch access permission

11.6.5.3 Status Register

The SPARC V8 architecture allows for a number of types of memory access error to be trapped. These trap types and trap handling in general are described in chapter 7 of the SPARC architecture manual [32]. According to the SPARC architecture manual the processor will automatically move to the next register window (i.e. it decrements the current window pointer) and copies the program counters (PC and nPC) to two local registers in the new window. The supervisor bit in the PSR is also set and the PSR can be saved to another local register by the trap handler (this does not happen automatically in hardware).

At the time of writing it is not clear whether the LEON core can easily accept memory access error trap types (i.e. the 8-bit *tt* field of the Trap Base register). Further investigation is needed to determine it this is possible and if existing trap types will cover the different types of bus error possible on SoPEC. Up to 32 implementation specific trap types are allowed so conditions unique to SoPEC can be handled in this manner.

If it is not possible for sufficient information about the cause of the bus error to be passed to the LEON core using the above mechanisms then a status register will be implemented to record the relevant information.

11.6.6 MMU Sub-block partition

As can be seen from Figure 19 and Figure 20 the MMU consists of five principal sub-blocks. For clarity the connections between these sub-blocks and other SoPEC blocks and between each of the sub-blocks are shown in two separate diagrams.
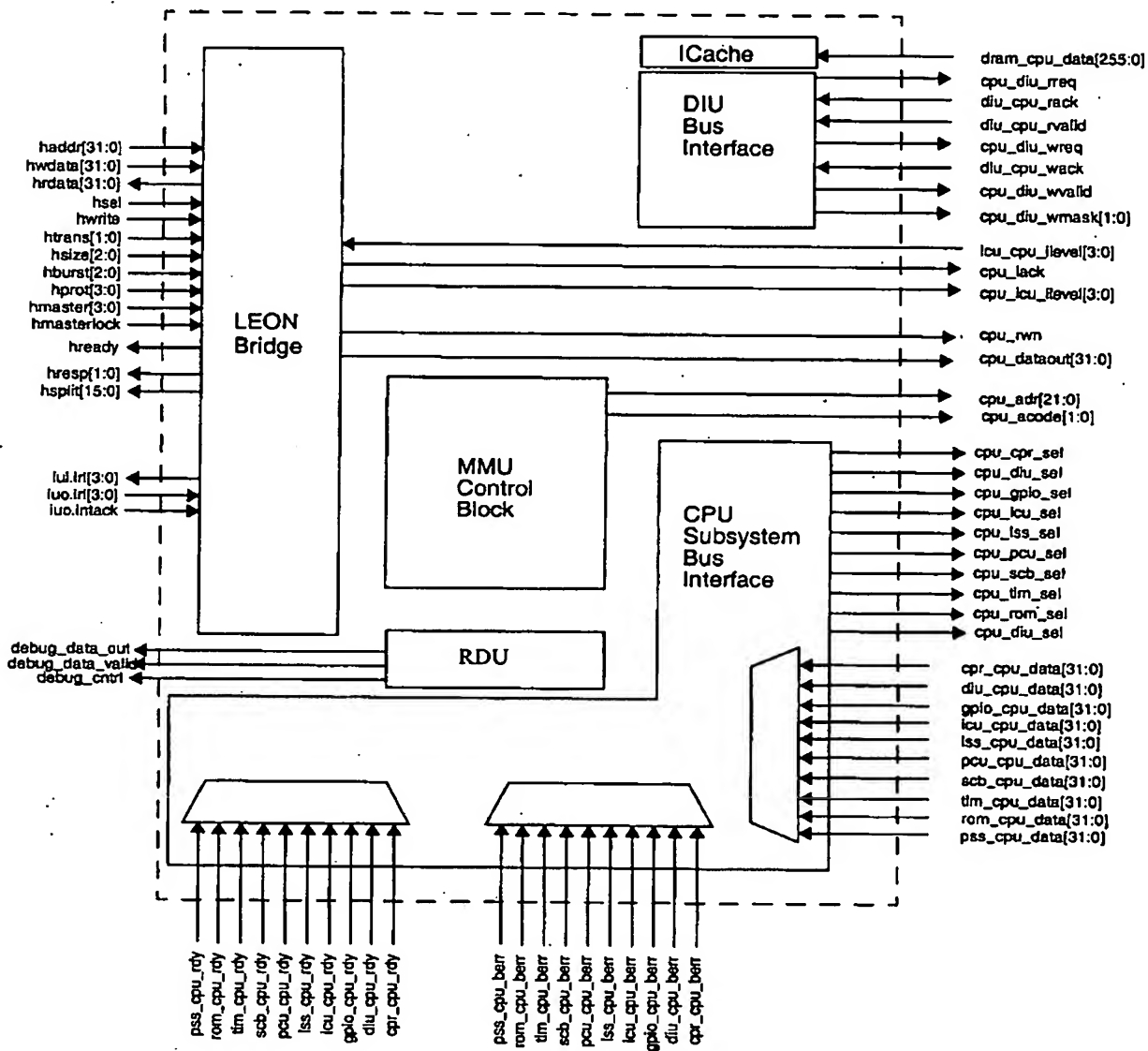
Figure 19. MMU Sub-block partition, external signal view
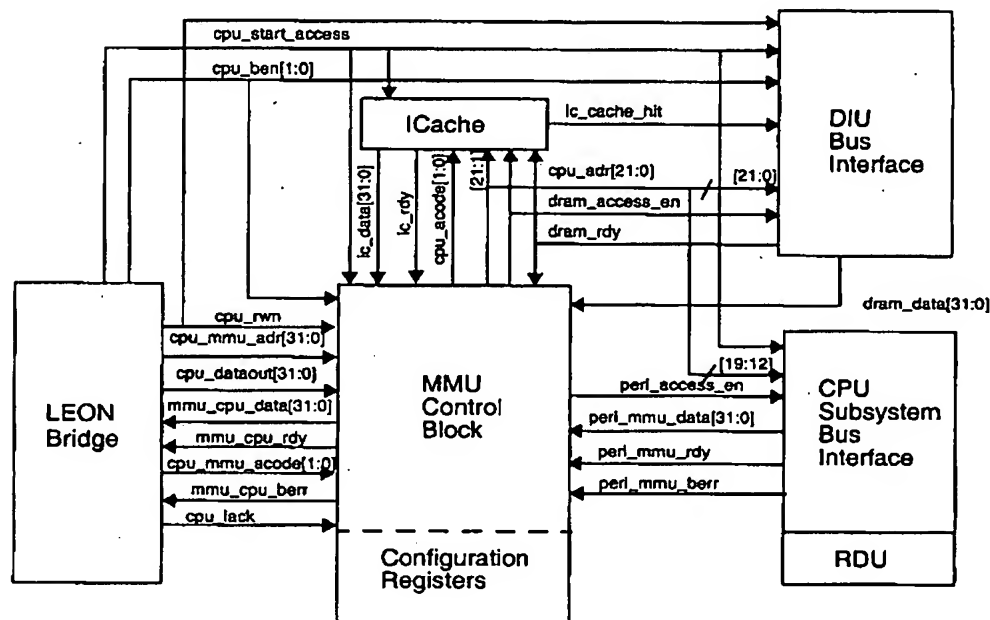
Figure 20. MMU Sub-block partition, internal signal view

11.6.6.1 LEON Bridge

At the time of writing it is expected that the LEON core will be used with its AHB interface rather than be modified to comply with the protocols used on SoPEC, in particular the DIU protocol for DRAM access. The LEON bridge consists of an AHB bridge and some glue logic. The AHB bridge will convert between the AHB and the DIU and CPU subsystem bus protocols. The AHB bridge will always be a slave on the AHB. Glue logic will be required to assist with endianness coherency, interrupts and other miscellaneous signalling.

Table 20. LEON bridge I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
LEON Bridge to AHB signals			
haddr[31:0]	32	In	AHB address bus
hwdata[31:0]	32	In	AHB write data bus
hrdata[31:0]	32	Out	AHB read data bus
hsel	1	In	AHB slave select signal

Table 20. LEON bridge I/Os

Portname	Pins	I/O	Description
hwrite	1	In	AHB write signal: 1 - Write access 0 - Read access
htrans	2	In	Indicates the type of the current transfer: 00 - IDLE 01 - BUSY 10 - NONSEQ 11 - SEQ
hsize	3	In	Indicates the size of the current transfer: 000 - Byte transfer 001 - Halfword transfer 010 - Word transfer 011 - 64-bit transfer (unsupported?) 1xx - Unsupported larger wordsizes
hburst	3	In	Indicates if the current transfer forms part of a burst and the type of burst: 000 - SINGLE 001 - INCR 010 - WRAP4 011 - INCR4 100 - WRAP8 101 - INCR8 110 - WRAP16 111 - INCR16
hprot	4	In	Protection control signals pertaining to the current access: hprot[0] - Opcode(0) / Data(1) access hprot[1] - User(0) / Supervisor access hprot[2] - Non-bufferable(0) / Bufferable(1) access (unsupported) hprot[3] - Non-cacheable(0) / Cacheable access
hmaster	4	In	Indicates the identity of the current bus master. This will always be the LEON core.
hmastlock	1	In	Indicates that the current master is performing a locked sequence of transfers.
hready	1	Out	Active high ready signal indicating the access has completed
hresp	2	Out	Indicates the status of the transfer: 00 - OKAY 01 - ERROR 10 - RETRY 11 - SPLIT
hsplit	16	Out	This 16-bit split bus is used by a slave to indicate to the arbiter which bus masters should be allowed attempt a split transaction. This feature will be unsupported on the AHB bridge
Toplevel/ Common LEON bridge signals			
cpu_dataout[31:0]	32	Out	Data out bus to both DRAM and peripheral devices.
cpu_rwn	1	Out	Read/NotWrite signal. 1 = Current access is a read access, 0 = Current access is a write access
icu_cpu_ilevel[3:0]	4	In	An interrupt is asserted by driving the appropriate priority level on *icu_cpu_ilevel*. These signals must remain asserted until the CPU executes an interrupt acknowledge cycle.
cpu_icu_ilevel[3:0]	4	In	Indicates the level of the interrupt the CPU is acknowledging when *cpu_iack* is high

Table 20. LEON bridge I/Os

Port name	Pins	I/O	Description
cpu_iack	1	Out	Interrupt acknowledge signal. The exact timing depends on the CPU core implementation
cpu_start_access	1	Out	Start Access signal indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_ben[1:0]	2	Out	Byte enable signals.
LEON core to LEON bridge signals			
iul.irl	4	Out	Interrupt level request to the LEON Integer Unit
iuo.irl	4	In	Acknowledged interrupt level from the LEON Integer Unit
iuo.intack	1	In	Interrupt acknowledge signal from the LEON Integer Unit
LEON bridge to MMU Control Block signals			
cpu_mmu_adr	32	Out	CPU Address Bus.
mmu_cpu_data	32	In	Data bus from the MMU
mmu_cpu_rdy	1	In	Ready signal from the MMU
cpu_mmu_acode	2	Out	Access code signals to the MMU
mmu_cpu_berr	1	In	Bus error signal from the MMU

Description:

The LEON bridge must ensure that all CPU bus and interrupt transactions are functionally correct and that the timing requirements are met. This sub-block is also responsible for ensuring endianness coherency i.e. guaranteeing that the correct data appears in the correct position on the data buses (*hrdata, cpu_dataout* and *mmu_cpu_data*) for every type of access. This is a requirement because the LEON uses big-endian addressing while the rest of SoPEC is little-endian.

It is expected that some signals (especially those external to the CPU block) will need to be registered here to meet the timing requirements. Careful thought will be required to ensure that overall CPU access times are not excessively degraded by the use of too many register stages.

11.6.6.2 DIU Bus Interface

The DIU bus interface will handle all valid accesses to the embedded DRAM via the DIU. The DIU bus interface ensures that the access conforms to the DIU bus protocol while the DIU manages the arbitration and data alignment.

Table 21. DIU Bus Interface I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common DIU Bus Interface signals			
dram_cpu_data[255:0]	256	In	Read data from the DRAM.
cpu_diu_rreq	1	Out	Read request to the DIU DRAM
diu_cpu_rack	1	In	Acknowledge from DIU that read request has been accepted.
diu_cpu_rvalid	1	In	Signal from DIU indicating that valid read data is on the *dram_cpu_data* bus
cpu_diu_wreq	1	Out	Write request to the DIU

Table 21. DIU Bus Interface I/Os

Port name	Pins	I/O	Description
diu_cpu_wack	1	In	Acknowledge from the DIU that the write request has been accepted
cpu_diu_wvalid	1	Out	Signal from the CPU to the DIU indicating that the data currently on the *cpu_dataout* bus is valid
cpu_diu_wmask[1:0]	2	Out	Flag indicating format of CPU write to DRAM. These signals are directly derived from the *cpu_ben* signals *cpu_diu_wmask* = 00: 8-bit write *cpu_diu_wmask* = 01: 16-bit write *cpu_diu_wmask* = 10: 32-bit write *cpu_diu_wmask* = 11: reserved *cpu_adr[2:0]* are driven in accordance with the width of the data access indicated by *cpu_diu_wmask*. Addresses cannot cross a 256-bit word DRAM boundary.
dram_rdy	1	Out	Data Ready signal. Indicates the data on the *dram_cpu_data* bus is valid for a read cycle or that the data was successfully dispatched to the DIU for a write cycle.
DIU Bus Interface to MMU Control Block signals			
cpu_adr[21:0]	22	In	Toplevel CPU Address bus.
dram_data[31:0]	32	Out	Data bus containing the 32 bits addressed by *cpu_adr[4:2]* from the 256-bit DRAM read bus *dram_cpu_data*
dram_access_en	1	In	Enable Access signal. A DRAM access cannot be initiated unless it has been enabled by the MMU Control Unit
DIU Bus Interface to ICache signals			
ic_cache_hit	1	In	Cache hit signal from the ICache. This indicates that the current CPU read request is being serviced by the ICache and so should not be retrieved from the DRAM.
DIU Bus Interface to LEON bridge signals			
cpu_ben[1:0]	2	In	Byte enable signals from the LEON bridge. These are forwarded on to the DIU as the *cpu_diu_wmask* signals
cpu_start_access	1	In	Start Access signal from the LEON bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.

Description:

The DIU Bus Interface handles all data transfers between the CPU (or ICache) and the DIU. This involves translating between the different protocols used on the DIU and CPU buses. The validity (i.e. is the CPU running in the correct mode for the address space being accessed) of an access is determined by the MMU Control Block which also checks that a DRAM access does not cross a 256-bit boundary (as required by the DIU) and the *dram_access_en* is asserted if it is a valid access. Invalid accesses do not initiate DRAM accesses. The operation of the DIU Bus Interface is described by the state machine shown in Figure 21 and the DIU bus protocol is described in more detail in section 20.9. The DIU will return a 256-bit dataword on *dram_cpu_data[255:0]* for every read access. The DIU Bus Interface must select the appropriate 32-bit word from this according to the word address given by *cpu_adr[4:2]*.
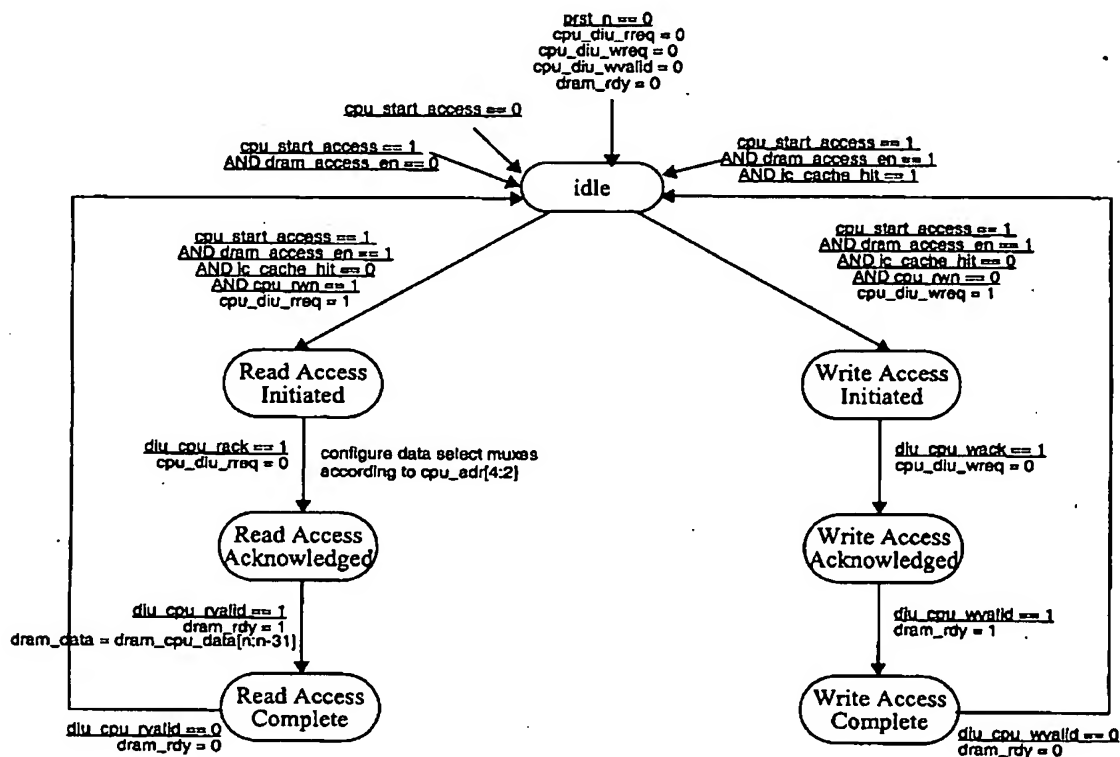
Figure 21. DIU Bus Interface state machine

11.6.6.3 CPU Subsystem Bus Interface

The CPU Subsystem Interface block handles all valid accesses to the peripheral blocks that comprise the CPU Subsystem.

Table 22. CPU Subsystem Bus Interface I/Os

Port name	#pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common CPU Subsystem Bus Interface signals			
cpu_cpr_sel	1	Out	CPR block select.
cpu_gpio_sel	1	Out	GPIO block select.
cpu_icu_sel	1	Out	ICU block select.
cpu_lss_sel	1	Out	LSS block select.
cpu_pcu_sel	1	Out	PCU block select.

Table 22. CPU Subsystem Bus Interface I/Os

Port name	Pins	In/Out	Description
cpu_scb_sel	1	Out	SCB block select.
cpu_tim_sel	1	Out	Timers block select.
cpu_rom_sel	1	Out	ROM block select.
cpu_pss_sel	1	Out	PSS block select.
cpu_diu_sel	1	Out	DIU block select.
cpr_cpu_data[31:0]	32	In	Read data bus from the CPR block
gpio_cpu_data[31:0]	32	In	Read data bus from the GPIO block
icu_cpu_data[31:0]	32	In	Read data bus from the ICU block
lss_cpu_data[31:0]	32	In	Read data bus from the LSS block
pcu_cpu_data[31:0]	32	In	Read data bus from the PCU block
scb_cpu_data[31:0]	32	In	Read data bus from the SCB block
tim_cpu_data[31:0]	32	In	Read data bus from the Timers block
rom_cpu_data[31:0]	32	In	Read data bus from the ROM block
pss_cpu_data[31:0]	32	In	Read data bus from the PSS block
diu_cpu_data[31:0]	32	In	Read data bus from the DIU block
cpr_cpu_rdy	1	In	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the CPR block and for a read cycle this means the data on *cpr_cpu_data* is valid.
gpio_cpu_rdy	1	In	GPIO ready signal to the CPU.
icu_cpu_rdy	1	In	ICU ready signal to the CPU.
lss_cpu_rdy	1	In	LSS ready signal to the CPU.
pcu_cpu_rdy	1	In	PCU ready signal to the CPU.
scb_cpu_rdy	1	In	SCB ready signal to the CPU.
tim_cpu_rdy	1	In	Timers block ready signal to the CPU.
rom_cpu_rdy	1	In	ROM block ready signal to the CPU.
pss_cpu_rdy	1	In	PSS block ready signal to the CPU.
diu_cpu_rdy	1	In	DIU register block ready signal to the CPU.
cpr_cpu_berr	1	In	Bus Error signal from the CPR block
gpio_cpu_berr	1	In	Bus Error signal from the GPIO block
icu_cpu_berr	1	In	Bus Error signal from the ICU block
lss_cpu_berr	1	In	Bus Error signal from the LSS block
pcu_cpu_berr	1	In	Bus Error signal from the PCU block
scb_cpu_berr	1	In	Bus Error signal from the SCB block
tim_cpu_berr	1	In	Bus Error signal from the Timers block
rom_cpu_berr	1	In	Bus Error signal from the ROM block
pss_cpu_berr	1	In	Bus Error signal from the PSS block
diu_cpu_berr	1	In	Bus Error signal from the DIU block
CPU Subsystem Bus Interface to MMU Control Block signals			
cpu_adr[19:12]	8	In	Toplevel CPU Address bus. Only bits 19-12 are required to decode the peripherals address space
peri_access_en	1	In	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit

Table 22. CPU Subsystem Bus Interface I/Os

Port name	Pins	I/O	Description
peri_mmu_data[31:0]	32	Out	Data bus from the selected peripheral
peri_mmu_rdy	1	Out	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.
peri_mmu_berr	1	Out	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral
CPU Subsystem Bus Interface to LEON bridge signals			
cpu_start_access	1	In	Start Access signal from the LEON bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.

Description:

The CPU Subsystem Bus Interface block performs simple address decoding to select a peripheral and multiplexing of the returned signals from the various peripheral blocks. The base addresses used for the decode operation are defined in Table 16. Note that access to the MMU configuration registers are handled by the MMU Control Block rather than the CPU Subsystem Bus Interface block. The CPU Subsystem Bus Interface block operation is described by the following pseudocode:

```
masked_cpu_adr = cpu_adr[19:12]
case (masked_cpu_adr)
   when TIM_base[19:12]
      cpu_tim_sel = peri_access_en      // The peri_access_en signal will have the
      peri_mmu_data = tim_cpu_data      // timing required for block selects
      peri_mmu_rdy = tim_cpu_rdy
      peri_mmu_berr = tim_cpu_berr
      all_other_selects = 0    //  Shorthand to ensure other cpu_block_sel signals
                               //  remain deasserted
   when LSS_base[19:12]
      cpu_lss_sel = peri_access_en
      peri_mmu_data = lss_cpu_data
      peri_mmu_rdy = lss_cpu_rdy
      peri_mmu_berr = lss_cpu_berr
      all_other_selects = 0
   when GPIO_base[19:12]
      cpu_gpio_sel = peri_access_en
      peri_mmu_data = gpio_cpu_data
      peri_mmu_rdy = gpio_cpu_rdy
      peri_mmu_berr = gpio_cpu_berr
      all_other_selects = 0
   when SCB_base[19:12]
      cpu_scb_sel = peri_access_en
      peri_mmu_data = scb_cpu_data
      peri_mmu_rdy = scb_cpu_rdy
      peri_mmu_berr = scb_cpu_berr
      all_other_selects = 0
   when ICU_base[19:12]
      cpu_icu_sel = peri_access_en
      peri_mmu_data = icu_cpu_data
      peri_mmu_rdy = icu_cpu_rdy
      peri_mmu_berr = icu_cpu_berr
      all_other_selects = 0
   when CPR_base[19:12]
      cpu_cpr_sel = peri_access_en
      peri_mmu_data = cpr_cpu_data
```

```
            peri_mmu_rdy = cpr_cpu_rdy
            peri_mmu_berr = cpr_cpu_berr
            all_other_selects = 0
      when ROM_base[19:12]
            cpu_rom_sel = peri_access_en
            peri_mmu_data = rom_cpu_data
            peri_mmu_rdy = rom_cpu_rdy
            peri_mmu_berr = rom_cpu_berr
            all_other_selects = 0
      when PSS_base[19:12]
            cpu_pss_sel = peri_access_en
            peri_mmu_data = pss_cpu_data
            peri_mmu_rdy = pss_cpu_rdy
            peri_mmu_berr = pss_cpu_berr
            all_other_selects = 0
      when DIU_base[19:12]
            cpu_diu_sel = peri_access_en
            peri_mmu_data = diu_cpu_data
            peri_mmu_rdy = diu_cpu_rdy
            peri_mmu_berr = diu_cpu_berr
            all_other_selects = 0
      when PCU_base[19:12]
            cpu_diu_sel = peri_access_en
            peri_mmu_data = pcu_cpu_data
            peri_mmu_rdy = pcu_cpu_rdy
            peri_mmu_berr = pcu_cpu_berr
            all_other_selects = 0
      when others
            all_block_selects = 0
            peri_mmu_data = 0x00000000
            peri_mmu_rdy = 0
            peri_mmu_berr = 1
      end case
```

11.6.6.4 MMU Control Block

The MMU Control Block determines whether every CPU access is a valid access. No more than one cycle is to be consumed in determining the validity of an access and all accesses must terminate with the assertion of either *mmu_cpu_rdy* or *mmu_cpu_berr*. To safeguard against stalling the CPU a simple bus timeout mechanism will be supported.

Table 23. MMU Control Block I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel/Common MMU Control Block signals			
cpu_adr[21:0]	22	Out	Address bus for both DRAM and peripheral access.
cpu_acode[1:0]	2	Out	CPU access code signals (*cpu_mmu_acode*) retimed to meet the CPU Subsystem Bus timing requirements
dram_access_en	1	Out	DRAM Access Enable signal. Indicates that the current CPU access is a valid DRAM access.
MMU Control Block to LEON bridge signals			

Table 23. MMU Control Block I/Os

Port name	Pins	I/O	Description
cpu_mmu_adr[31:0]	32	In	CPU core address bus.
cpu_dataout[31:0]	32	In	Toplevel CPU data bus
mmu_cpu_data[31:0]	32	Out	Data bus to the CPU core. Carries the data for all CPU read operations
cpu_rwn	1	In	Toplevel CPU Read/notWrite signal.
cpu_mmu_acode[1:0]	2	In	CPU access code signals
mmu_cpu_rdy	1	Out	Ready signal to the CPU core. Indicates the completion of all valid CPU accesses.
mmu_cpu_berr	1	Out	Bus Error signal to the CPU core. This signal is asserted to terminate an invalid access.
cpu_start_access	1	In	Start Access signal from the LEON bridge indicating the start of a data transfer and that the *cpu_adr, cpu_dataout, cpu_rwn* and *cpu_acode* signals are all valid. This signal is only asserted during the first cycle of an access.
cpu_iack	1	In	Interrupt Acknowledge signal from the CPU. This signal is only asserted during an interrupt acknowledge cycle.
cpu_ben[1:0]	2	In	Byte enable signals indicating which bytes of the 32-bit bus are being accessed.
MMU Control Block to DIU Bus interface signals			
dram_rdy	1	In	Data Ready signal. Indicates the data on the *dram_cpu_data* bus is valid for a read cycle or that the data was successfully dispatched to the DIU for a write cycle.
MMU Control Block to ICache signals			
ic_data[31:0]	32	In	Data bus from the ICache
ic_rdy	1	In	Ready signal from the ICache indicating the data on *ic_data* is valid
MMU Control Block to CPU Subsystem Bus interface signals			
peri_access_en	1	Out	Enable Access signal. A peripheral access cannot be initiated unless it has been enabled by the MMU Control Unit
peri_mmu_data[31:0]	32	In	Data bus from the selected peripheral
peri_mmu_rdy	1	In	Data Ready signal. Indicates the data on the *peri_mmu_data* bus is valid for a read cycle or that the data was successfully written to the peripheral for a write cycle.
peri_mmu_berr	1	In	Bus Error signal. Indicates a bus error has occurred in accessing the selected peripheral

Description:

The MMU Control Block is responsible for the MMU's core functionality, namely determining whether or not an access to any part of the address map is valid. An access is considered valid if it is to a mapped area of the address space and if the CPU is running in the appropriate mode for that address space. Furthermore the MMU control block must correctly handle the special cases that are: an interrupt acknowledge cycle, a reset exception vector fetch, an access that crosses a 256-bit DRAM word boundary and a bus timeout condition. The following pseudocode shows the logic required to implement the MMU Control Block functionality. It does not deal with the timing relationships of the various signals - it is the designer's responsibility to ensure that these relationships are correct and comply with the different bus protocols. For simplicity the pseudocode is split up into numbered sections so that the functionality may be seen more easily.

PS0 Description: This first segment of code defines a number of constants and variables that are used elsewhere in this description. Most signals have been defined in the I/O descriptions of the MMU sub-blocks that precede this section of the document. The *post_reset_state* variable is used later (in section PS4) to determine if we should translate the reset exception vector address or trap a null pointer access.

PS0:

```
   const UnusedBottom = 0x002AC000
   const DRAMTop = 0x0027FFFF
   const UserDataSpace = b01
   const UserProgramSpace = b00
   const SupervisorDataSpace = b11
   const SupervisorProgramSpace = b10
   const timeout_limit = 0x40   // Need to confirm that this is a suitable value
   const ResetExceptionCycles = 0x8

   cpu_adr_peri_masked[7:0] = cpu_mmu_adr[19:12]
   cpu_adr_dram_masked[16:0] = cpu_mmu_adr & 0x003FFFE0

   if (prst_n == 0) then       // Initialise everything
      cpu_adr = cpu_mmu_adr[21:0]
      peri_access_en = 0
      dram_access_en = 0
      mmu_cpu_data = peri_mmu_data
      mmu_cpu_rdy = 0
      mmu_cpu_berr = 0
      post_reset_state = TRUE
      access_initiated = FALSE
      cpu_access_cnt = 0

   // The following is used to determine if we are coming out of reset for the purposes of
   // reset exception vector redirection. There may be a convenient signal in the CPU core
   // that we could use instead of this.
   if ((cpu_start_access == 1) AND (cpu_access_cnt < ResetExceptionCycles ) AND
       (clock_tick == TRUE)) then
      cpu_access_cnt = cpu_access_cnt +1
   else
      post_reset_state = FALSE
```

PS1 Description: This section is at the top of the hierarchy that determines the validity of an access. The address is tested to see which macro-region (i.e. Unused, CPU Subsystem or DRAM) it falls into or whether the reset exception vector is being accessed.

PS1:

```
   if (cpu_mmu_adr >= UnusedBottom) then
      // The access is to an invalid area of the address space. See section PS2

   elsif ((cpu_mmu_adr > DRAMTop) AND (cpu_mmu_adr < UnusedBottom)) then
      // We are in the CPU Subsystem/PEP Subsystem address space. See section PS3

   // Only remaining possibility is an access to DRAM address space
   // First we need to intercept the special case for the reset exception vector

   elsif (cpu_mmu_adr < 0x00000010) then
      // The reset exception is being accessed. See section PS4

   elsif ((cpu_adr_dram_masked >= Region0Bottom) AND (cpu_adr_dram_masked <=
            Region0Top) ) then
         // We are in Region0. See section PS5

   elsif ((cpu_adr_dram_masked >= RegionNBottom) AND (cpu_adr_dram_masked <=
```

```
                        RegionNTop) ) then   // we are in RegionN
                        // Repeat the Region0 (i.e. section PS5) logic for each of Region1 to Region7

             else  // We could end up here if there were gaps in the DRAM regions
                peri_access_en = 0
                dram_access_en = 0
                mmu_cpu_berr = 1      // we have an unknown access error, most likely due to hitting
                mmu_cpu_rdy = 0       // a gap in the DRAM regions

             // Only thing remaining is to implement a bus timeout function. This is done in PS6

             end
```

PS2 Description: Accesses to the large unused area of the address space are trapped by this section. No bus transactions are initiated and the *mmu_cpu_berr* signal is asserted.

PS2:
```
        elsif (cpu_mmu_adr >= UnusedBottom) then
             peri_access_en = 0   // The access is to an invalid area of the address space
             dram_access_en = 0
             mmu_cpu_berr = 1
             mmu_cpu_rdy = 0
```

PS3 Description: This section deals with accesses to CPU Subsystem peripherals, including the MMU itself. If the MMU registers are being accessed then no external bus transactions are required. Access to the MMU registers is only permitted of the CPU is making a data access from supervisor mode, otherwise a bus error is asserted and the access terminated. For non-MMU accesses then transactions occur over the CPU Subsystem Bus and each peripheral is responsible for determining whether or not the CPU is in the correct mode (based on the *cpu_acode* signals) to be permitted access to its registers. Note that all of the PEP registers are accessed via the PCU which is on the CPU Subsystem Bus.

PS3:
```
        elsif ((cpu_mmu_adr > DRAMTop) AND (cpu_mmu_adr < UnusedBottom)) then
             // We are in the CPU Subsystem/PEP Subsystem address space

             cpu_adr = cpu_mmu_adr[21:0]
             if (cpu_adr_peri_masked == MMU_base) then  // access is to local registers
                   peri_access_en = 0
                   dram_access_en = 0
                if (cpu_acode == SupervisorDataSpace) then
                     for (i=0; i<26; i++) {
                         if ((i == cpu_mmu_adr[6:2]) then   // selects the addressed register
                             if (cpu_rwn == 1) then
                                 mmu_cpu_data[16:0] = MMUReg[i]    // MMUReg[i] is one of the
                                 mmu_cpu_rdy = 1.                  // registers in Table 18
                                 mmu_cpu_berr = 0
                             else  // write cycle
                                 MMUReg[i] = cpu_dataout[16:0]
                                 mmu_cpu_rdy = 1
                                 mmu_cpu_berr = 0
                         else  // there is no register mapped to this address
                             mmu_cpu_berr = 1  // do we really want a bus_error here as registers
                             mmu_cpu_rdy = 0   // are just mirrored in other blocks

                else // we have an access violation
                    mmu_cpu_berr = 1
                    mmu_cpu_rdy = 0

             else // access is to something else on the CPU Subsystem Bus
```

```
                    peri_access_en = 1
                    dram_access_en = 0
                    mmu_cpu_data = peri_mmu_data
                    mmu_cpu_rdy = peri_mmu_rdy
                    mmu_cpu_berr = peri_mmu_berr
```

PS4 Description: The only correct accesses to the locations beneath 0x00000010 are fetches of the reset trap handling routine and these should be the first accesses after reset. Here we trap all other accesses to these locations regardless of the CPU mode. This most likely cause of such an access will be the use of a null pointer in the program executing on the CPU.

PS4:
```
    elsif (cpu_mmu_adr < 0x00000010) then    //may need to translate a wider range - depends
        if (post_reset_state == TRUE)) then  // on how LEON handles the reset exception.
            cpu_adr[21:0] = {ROM_base[21:3], cpu_mmu_adr[2:0]}
            peri_access_en = 1
            dram_access_en = 0
            mmu_cpu_data = peri_mmu_data
            mmu_cpu_rdy = peri_mmu_rdy
            mmu_cpu_berr = peri_mmu_berr
        else  // we have a problem (almost certainly a null pointer)
            peri_access_en = 0
            dram_access_en = 0
            mmu_cpu_berr = 1
            mmu_cpu_rdy = 0
```

PS5 Description: This large section of pseudocode simply checks whether the access is within the bounds of DRAM Region0 and if so whether or not the access is of a type permitted by the *Region0Control* register. If the access is permitted then a DRAM access is initiated for all data accesses and for instruction fetches that result in a cache miss. All instruction fetches are returned via the ICache interface regardless of whether they come from a cache hit or refill from DRAM. If the access is not of a type permitted by the *Region0Control* register then the access is terminated with a bus error.

PS5:
```
    elsif ((cpu_adr_dram_masked >= Region0Bottom) AND (cpu_adr_dram_masked <=
            Region0Top) ) then   // we are in Region0

            // We need to check that the DRAM access does not cross a 256-bit boundary
            // Only 16 or 32-bit CPU accesses are capable of traversing a 256-bit boundary

            if ( ((cpu_mmu_adr[4:0] == 0x1F) AND ((cpu_ben == b01) OR (cpu_ben == b10)))
                OR ((cpu_mmu_adr[4:0] == 0x1E) AND (cpu_ben == b10))
                OR ((cpu_mmu_adr[4:0] == 0x1D) AND (cpu_ben == b10)) ) then
                peri_access_en = 0
                dram_access_en = 0
                mmu_cpu_berr = 1
                mmu_cpu_rdy = 0

            else   // access does not cross 256-bit boundary so we can proceed
                cpu_adr = cpu_mmu_adr[21:0]
                if (cpu_rwn == 1) then
                    if ((cpu_acode == SupervisorProgramSpace AND Region0Control(2) == 1))
                        OR (cpu_acode == UserProgramSpace AND Region0Control(5) == 1)) then
                                        // this is a valid instruction fetch from Region0
                        peri_access_en = 0
                        dram_access_en = 1
                        mmu_cpu_data = ic_data
                        mmu_cpu_rdy = ic_rdy
```

```
                              mmu_cpu_berr = 0

            elsif ((cpu_acode == SupervisorDataSpace AND Region0Control[0] == 1)
               OR (cpu_acode == UserDataSpace AND Region0Control[3] == 1)) then
                                         // this is a valid read access from Region0
                  peri_access_en = 0
                  dram_access_en = 1
                  mmu_cpu_data = dram_data // possibly drc_data if dcache is used
                  mmu_cpu_rdy = dram_rdy   // possibly drc_rdy
                  mmu_cpu_berr = 0

            else                          // we have an access violation
                  peri_access_en = 0
                  dram_access_en = 0
                  mmu_cpu_berr = 1
                  mmu_cpu_rdy = 0

         else                             // it is a write access
            if ((cpu_acode == SupervisorDataSpace AND Region0Control[1] == 1)
               OR (cpu_acode == UserDataSpace AND Region0Control[4] == 1)) then
                                         // this is a valid write access to Region0
                  peri_access_en = 0
                  dram_access_en = 1
                  mmu_cpu_rdy = dram_rdy  // possibly dwc_rdy if dcache is used
                  mmu_cpu_berr = 0
            else                          // we have an access violation
                  peri_access_en = 0
                  dram_access_en = 0
                  mmu_cpu_berr = 1
                  mmu_cpu_rdy = 0
```

PS6 Description: This final section of pseudocode deals with the special case of a bus timeout. This occurs when an access has been initiated but has not completed before the *timeout_limit* number of *pclk* cycles. While access to both DRAM and CPU/PEP Subsystem registers will take a variable number of cycles (due to DRAM traffic, PCU command execution or the different timing required to access registers in imported IP) each access should complete before the *timeout_limit* occurs. Therefore it should not be possible to stall the CPU by locking either the CPU Subsystem or DIU buses. However given the fatal effect such a stall would have it is considered prudent to implement bus timeout detection.

PS6:
```
   // Only thing remaining is to implement a bus timeout function.

   if ((cpu_start_access == 1) then
      access_initiated = TRUE
      timeout_countdown = BusTimeout

   if ((mmu_cpu_rdy == 1 ) OR (mmu_cpu_berr ==1 )) then
      access_initiated = FALSE
      peri_access_en = 0
      dram_access_en = 0

   if ((clock_tick == TRUE) AND (access_initiated == TRUE))
      if (timeout_countdown > 0) then
         timeout_countdown--
      else  // timeout has occurred
         peri_access_en = 0          // abort the access
         dram_access_en = 0
         mmu_cpu_berr = 1
         mmu_cpu_rdy = 0
```

11.6.6.5 ICache

The ICache sub-block implementation is described in section 11.7.1.1.

11.7 CACHE

The decision on what type of caching solution to use on SoPEC is still open for the moment. There are two probable solutions: a) use the LEON caches with a minimal configuration (1 KB I and D caches) and b) use separate, simple one line 256-bit caches for instruction, data read and data write accesses. From a performance and (most likely) implementation point of view the LEON caches are the best solution however they are much bigger than the one line caches (approx 6x). The one line caches do not offer the same degree of performance improvement as the LEON caches and are likely to add an extra cycle to all memory accesses. The performance penalty for a LEON cache miss (i.e. for all memory accesses if we are not using the LEON caches) and the the best and worst case access times from DRAM have yet to be fully determined. The final decision on which caching solution to use will be made when all such information is available.

Therefore the section on caches, which was present in previous versions of this document but is now mostly out of date, has been removed (the ICache is still relevant if one line caches are used and so is retained).

11.7.1 Instruction Cache

A caching mechanism would offer the advantage of greater aggregate performance while still guaranteeing a minimum level of performance. While greater performance may not be required at present for this application the caching mechanism offers greater efficiency (i.e. MIPS/MHz) and so the CPU clock could be reduced without affecting, or only negligibly affecting, the operating performance. The advantage here is that the design is scalable - better performance can be achieved by simply increasing the clock rate.

As all reads from the embedded DRAM on SoPEC produce words that are 256 bits wide it is inefficient to hook this up to a 32-bit CPU bus as 224 bits of each read would be discarded. If the full 256-bit word is stored locally to the CPU as a single-line cache then a ??x performance improvement could be obtained in the typical case (this is of course highly code dependent). This single line cache would be very easy to implement as it would just involve the address to be compared to a single tag and no replacement algorithm would be required. Furthermore the area impact would be minor and there should be no performance penalty for cache misses. As the *dram_cpu_data* bus is 256 bits wide the requested word is immediately available to the CPU i.e. we do not need to perform critical word first reordering of the data.

The instruction cache is only accessed for instruction fetches, not all CPU reads. These can be differentiated by signals emanating from the CPU. Non-instruction CPU reads would be supported by the data cache. In the case of a cache miss the read request is processed by the MMU to ensure the request is valid before a read request is generated on the relevant external (to the CPU block) bus. The MMU should be informed of a cache hit to ensure it does not generate an unneccessary read request. This requires that the regions used to store code are aligned on 32-byte (256-bit) boundaries.

As there is no requirement to have more time deterministic code execution the instruction cache cannot be disabled.

11.7.1.1 ICache Implementation

The Instruction Cache used in SoPEC is capable of storing just a single 256-bit DRAM word. An implementation is depicted in Figure 22 below. The block I/Os are given in Table 24 and these should be viewed in conjunction with Figure 19 and Figure 20 for a complete depiction of the connectivity of the block.
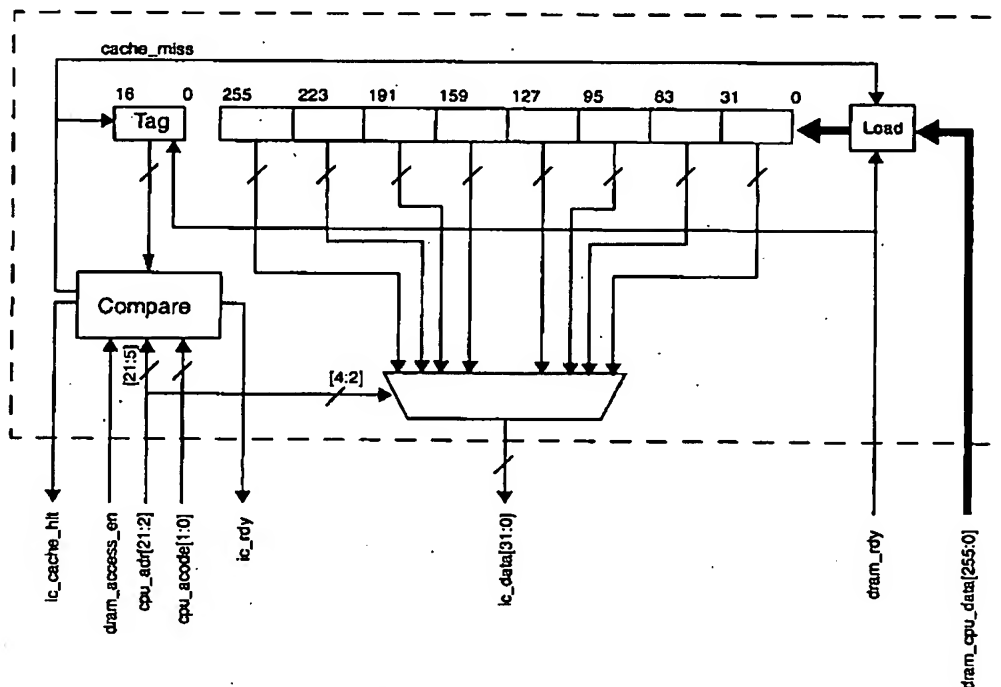
Figure 22. ICache Block Diagram

Table 24. ICache I/Os

Port name	Pins	I/O	Description
Global SoPEC signals			
prst_n	1	In	Global reset. Synchronous to *pclk*, active low.
pclk	1	In	Global clock
Toplevel ICache signals			
dram_cpu_data[255:0]	256	In	Data bus from the DIU
cpu_acode[1:0]	2	In	CPU access control signals
cpu_adr[21:2]	20	In	CPU core address bus.
ICache to DIU Bus Interface signals			
ic_cache_hit	1	Out	Cache hit signal. This indicates that the current CPU read request is being serviced by the ICache and so should not be retrieved from the DRAM.
dram_rdy	1	In	Data Ready signal. Indicates the data on the *dram_cpu_data* bus is valid.
ICache to MMU Control Block signals			
ic_data[31:0]	32	Out	ICache data bus

Table 24. ICache I/Os

Port name	Pins	I/O	Description
ic_rdy	1	Out	Ready signal from the ICache indicating the data on *ic_data* is valid
dram_access_en	1	Out	DRAM access enable signal. Indicates that the current CPU access is a valid DRAM access.

Description:

The Tag stores the DRAM word address of the word currently in cache. The Tag contents are compared with *cpu_adr[21:5]* each time the CPU requests an instruction fetch from a valid DRAM address (indicated by *cpu_acode[0]* and *dram_access_en*). If a match occurs (i.e. a cache hit) the access is serviced by returning the correct 32 bits (as selected by cpu_adr[4:2]) to the MMU Control Block. If a match does not occur (i.e. a cache miss) the *ic_cache_hit* line is held low indicating to the DIU Bus Interface that a DRAM access should commence. Completion of the DRAM access is signalled by the assertion of *dram_rdy* and this causes the ICache contents to be updated, the Tag value replaced and the relevant 32 bits forwarded to the CPU accompanied by the assertion of the *ic_rdy* signal. It is updated each time the cache line is refilled from DRAM. All instruction fetches from DRAM are cacheable, regardless of which DRAM region is being accessed (although the access permissions still need to match those programmed for the region) and whether the CPU is in user or supervisor mode.

11.7.2 Data Cache

11.8 REALTIME DEBUG UNIT (RDU)

The RDU facilitates the observation of the contents of most of the CPU addressable registers in the SoPEC device in addition to some pseudo-registers in realtime. The contents of pseudo-registers, i.e. registers that are collections of otherwise unobservable signals and that do not affect the functionality of a circuit, are defined in each block as required. Many blocks do not have pseudo-registers and some blocks (e.g. ROM , PSS) do not make debug information available to the RDU as it would be of little value in realtime debug.

Each block that supports realtime debug observation features a *DebugSelect* register that controls a local mux to determine which register is output on the block's data bus (i.e. *block_cpu_data*). One small drawback with reusing the blocks data bus is that the debug data cannot be present on the same bus during a CPU read from the block. An accompanying active high *block_cpu_debug_valid* signal is used to indicate when the data bus contains valid debug data and when the bus is being used by the CPU. There is no arbitration for the bus as the CPU will always have access when required. A block diagram of the RDU is shown in Figure 23.
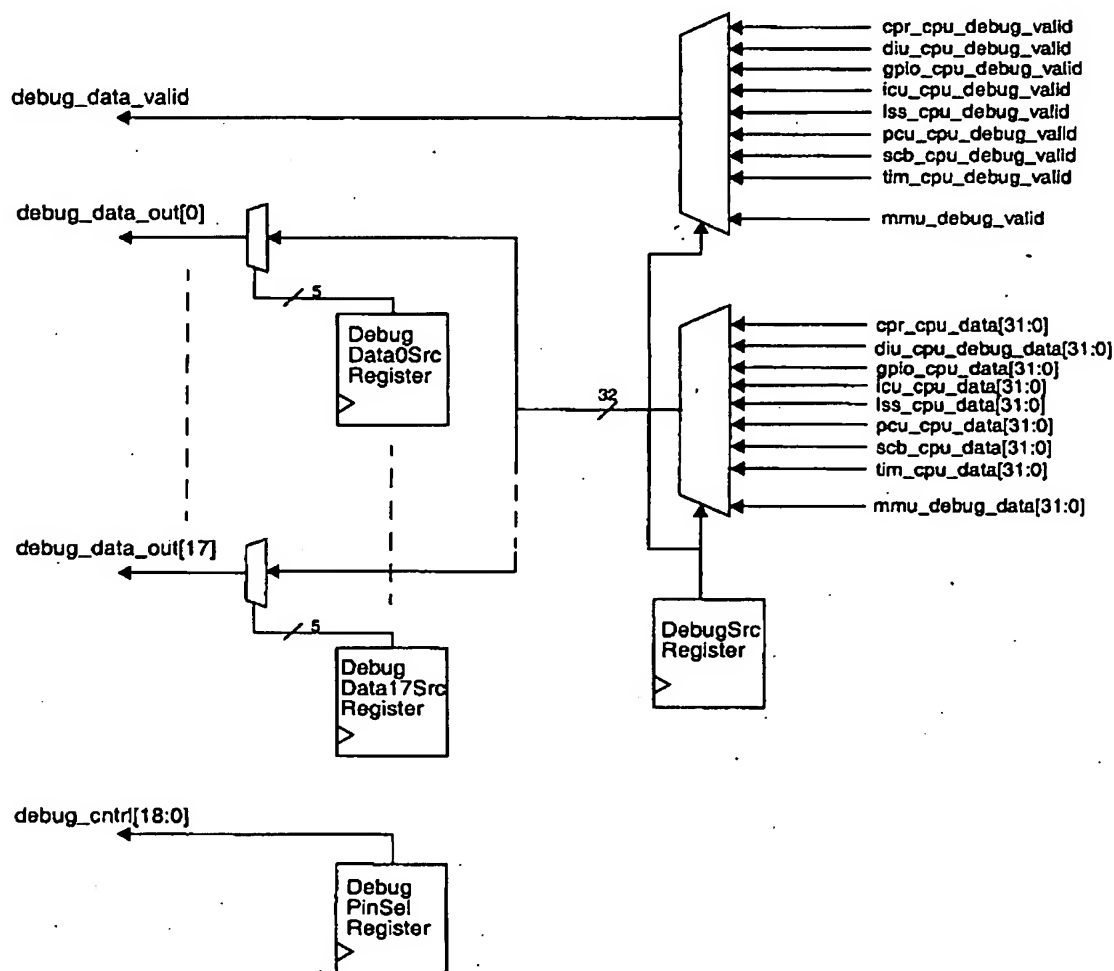
Figure 23. Realtime Debug Unit block diagram

Table 25. RDU I/Os

Portname	Pins	I/O	Description
diu_cpu_data	32	In	Read data bus from the DIU block
cpr_cpu_data	32	In	Read data bus from the CPR block
gpio_cpu_data	32	In	Read data bus from the GPIO block
icu_cpu_data	32	In	Read data bus from the ICU block
lss_cpu_data	32	In	Read data bus from the LSS block
pcu_cpu_debug_data	32	In	Read data bus from the PCU block

Table 25. RDU I/Os

Port name	Pins	I/O	Description
scb_cpu_data	32	In	Read data bus from the SCB block
tim_cpu_data	32	In	Read data bus from the TIM block
diu_cpu_debug_valid	1	In	Signal indicating the data on the *diu_cpu_data* bus is valid debug data.
tim_cpu_debug_valid	1	In	Signal indicating the data on the *tim_cpu_data* bus is valid debug data.
scb_cpu_debug_valid	1	In	Signal indicating the data on the *scb_cpu_data* bus is valid debug data.
pcu_cpu_debug_valid	1	In	Signal indicating the data on the *pcu_cpu_data* bus is valid debug data.
lss_cpu_debug_valid	1	In	Signal indicating the data on the *lss_cpu_data* bus is valid debug data.
icu_cpu_debug_valid	1	In	Signal indicating the data on the *icu_cpu_data* bus is valid debug data.
gpio_cpu_debug_valid	1	In	Signal indicating the data on the *gpio_cpu_data* bus is valid debug data.
cpr_cpu_debug_valid	1	In	Signal indicating the data on the *cpr_cpu_data* bus is valid debug data.
debug_data_out	18	Out	Output debug data to be muxed on to the PHI/GPIO/other pins
debug_data_valid	1	Out	Debug valid signal indicating the validity of the data on *debug_data_out*. This signal is used in all debug configurations
debug_cntrl	19	Out	Control signal for each PHI bound debug data line indicating whether or not the debug data should be selected by the pin mux

As there are no spare pins that can be used to output the debug data to an external capture device some of the existing I/Os will have a debug multiplexer placed in front of them to allow them be used as debug pins. Unfortunately many of the pins on SoPEC cannot even be multiplexed in this fashion so it will not be possible to output a full 32-bit debug data word every cycle. The exact number of pins available for multiplexing had yet to be finalised at the time of writing. This specification assumes 20 pins will be available but this can easily be revised up or, more likely, down. Furthermore not every pin that has a debug mux will always be available to carry the debug data as they may be engaged in their primary purpose e.g. as a GPIO pin. The RDU therefore outputs a *debug_cntrl* signal with each debug data bit to indicate whether the mux associated with each debug pin should select the debug data or the normal data for the pin.The *DebugPinSel* is used to determine which of the 20? potential debug pins are enabled for debug at any particular time.

As it is not possible to output a full 32-bit debug word every cycle the RDU supports the outputting of an n-bit sub-word every cycle to the enabled debug pins. Each debug test would then need to be re-run a number of times with a different portion of the debug word being output on the n-bit sub-word each time. The data from each run should then be correlated to create a full 32-bit (or whatever size is needed) debug word for every cycle. The *debug_data_valid* and *pclk_out* signals will accompany every sub-word to allow the data to be sampled correctly. The *pclk_out* signal is sourced close to its output pad rather than in the RDU to minimise the skew between the rising edge of the debug data signals (which should be registered close to their output pads) and the rising edge of *pclk_out*.

As multiple debug runs will be needed to obtain a complete set of debug data the n-bit sub-word will need to contain a different bit pattern for each run. For maximum flexibility each debug pin has an associated *DebugDataSrc* register that allows any of the 32 bits of the debug data word to be output on that particular

debug data pin. The debug data pin must be enabled for debug operation by having its corresponding bit in the *DebugPinSel* register set for the selected debug data bit to appear on the pin.

The size of the sub-word is determined by the number of enabled debug pins which is controlled by the *DebugPinSel* register. Note that the *debug_data_valid* signal is always output. Furthermore *debug_cntrl[0]* (which is configured by *DebugPinSel[0]*) controls the mux for both the *debug_data_valid* and *pclk_out* signals as both of these must be enabled for any debug operation.

The mapping of *debug_data_out[n]* signals onto individual pins will take place outside the RDU. When the exact mapping has been finalised it will be recorded here. A proposed mapping is shown in Table 26 below.

Table 26. Example DebugPinSel mapping

Bit	Pin
0	phi_frclk. The *debug_data_valid* signal will appear on this pin when enabled. Enabling this pin also automatically enables the phi_readl pin which will output the *pclk_out* signal
1	phi_profile
2	phi_isyncl
3	test pin1
4	test pin2
5-18	gpio[0...13]

Table 27. RDU Configuration Registers

| Address offset from MMU base	Register	#bits	Reset	Description
0x80	DebugSrc	4	0x00	Denotes which block is supplying the debug data. The encoding of this block is given below. 0 - MMU 1 - TIM 2 - LSS 3- GPIO 4 - SCB 5 - ICU 6 - CPR 7 - DIU 8 - PCU
0x84	DebugPinSel	19	0x0_0000	Determines whether a pin is used for debug data output. A provisional mapping of pin to bit position is given in Table 26. 1 - Pin outputs debug data 0 - Normal pin function
0x88 to 0xCC	DebugDataSrcN	5	0x00	Selects which bit of the 32-bit debug data word will be outputted on debug_data_out[N]

11.9 INTERRUPT OPERATION

The interrupt controller unit (see chapter 14) generates an interrupt request by driving interrupt request lines with the appropriate interrupt level. LEON supports 15 levels of interrupt with level 15 as the highest

level (the SPARC architecture manual [32] states that level 15 is non-maskable but we have the freedom to mask this if desired). The CPU will begin processing an interrupt exception when execution of the current instruction has completed and it will only do so if the interrupt level is higher than the current processor priority. If a second interrupt request arrives with the same level as an executing interrupt service routine then the exception will not be processed until the executing routine has completed.

When an interrupt trap occurs the LEON hardware will place the program counters (PC and nPC) into two local registers. The interrupt handler routine is expected, as a minimum, to place the PSR register in another local register to ensure that the LEON can correctly return to its pre-interrupt state. The 4-bit interrupt level (*irl*) is also written to the trap type (*tt*) field of the TBR (Trap Base Register) by hardware. The TBR then contains the vector of the trap handler routine the processor will then jump. The TBA (Trap Base Address) field of the TBR must have a valid value before any interrupt processing can occur so it should be configured at an early stage.

Interrupt pre-emption is supported while ET (Enable Traps) bit of the PSR is set. This bit is cleared during the initial trap processing. In initial simulations the ET bit was observed to be cleared for up to 30 cycles. This causes significant additional interrupt latency in the worst case where a higher priority interrupt arrives just as a lower priority one is taken.

The interrupt acknowledge cycles shown in Figure 24 below are derived from simulations of the LEON processor and accompanying interrupt controller. This interrupt controller will be replaced by the ICU in the SoPEC design. The LEON signal names are used for future reference. An interrupt is asserted by driving its (encoded) level on the *iui.irl[3:0]* signals. The LEON core responds to this, with variable timing, by reflecting the level of the taken interrupt on the *iuo.irl[3:0]* signals and asserting the acknowledge signal *iuo.intack*. The interrupt controller then removes the interrupt level one cycle after it has seen the level been acknowledged by the core. If there is another pending interrupt (of lower priority) then this should be driven on *iui.irl[3:0]* and the CPU will take that interrupt (the level 9 interrupt in the example below) once it has finished processing the higher priority interrupt. The *iuo.irl[3:0]* signals always reflect the level of the last taken interrupt, even when the CPU has finished processing all interrupts.

Figure 24. Interrupt acknowledge cycles for a single and pending interrupts
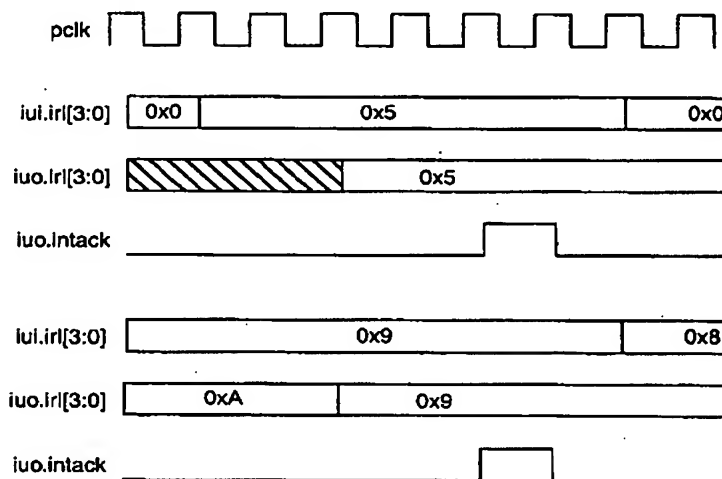
11.10 BOOT OPERATION

See section 17.2 for a description of the SoPEC boot operation.

11.11 SOFTWARE DEBUG

Software debug mechanisms are discussed in the "SoPEC Software Debug" document [15].

12 Serial Communications Block (SCB)

12.1 OVERVIEW

The Serial Communications Block (SCB) handles the movement of all data between the SoPEC and the host device (i.e. PC) and between master and slave SoPEC devices. The SCB consists of a USB1.1 device controller, an Inter-SoPEC Interface (ISI) and a DMA manager. A block diagram of the SCB is shown in Figure 25 below. The major blocks of the SCB, namely the ISI, USB and DMA manager, could be implemented as separate blocks but are integrated to take advantage of the performance gains and design simplifications that a tighter coupling allow.
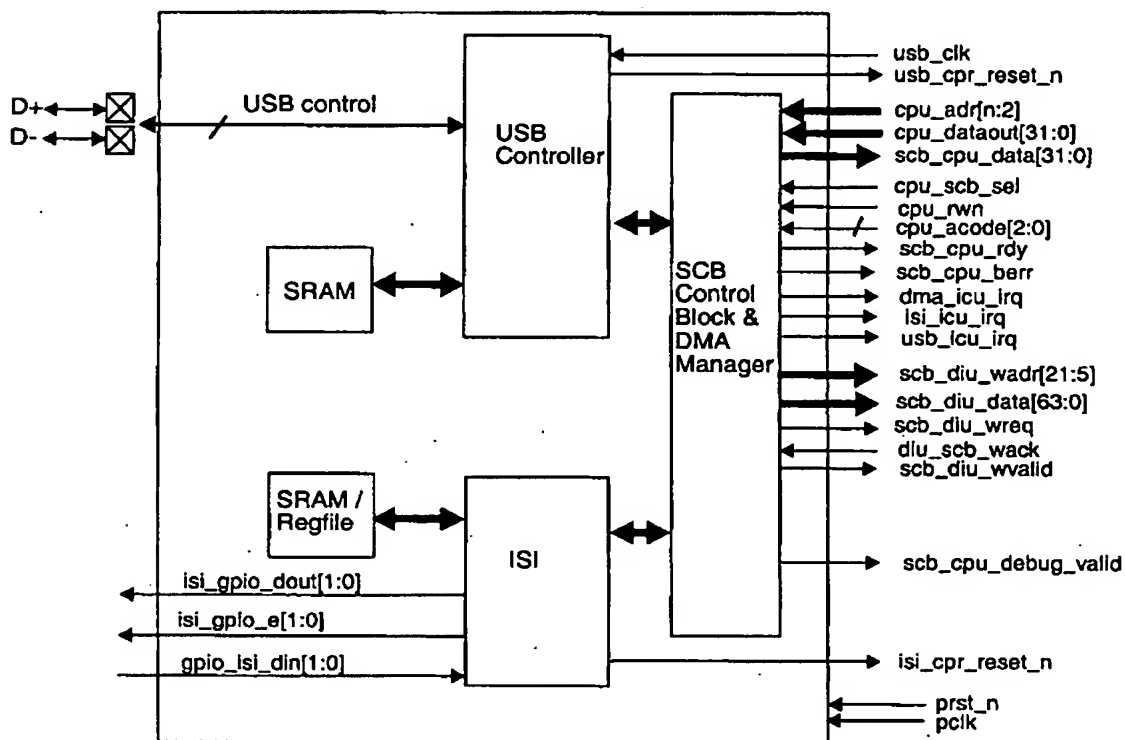
Figure 25. Serial Communications Block

The USB Controller will be an imported piece of IP. There are many possible sources of this block but it is likely that it will be supplied by the silicon vendor - all three current silicon vendor candidates will supply USB1.1 controllers, although some of these have been sourced from a third party.

The SCB can be seen in the context of the overall SoPEC device in Figure 26 below
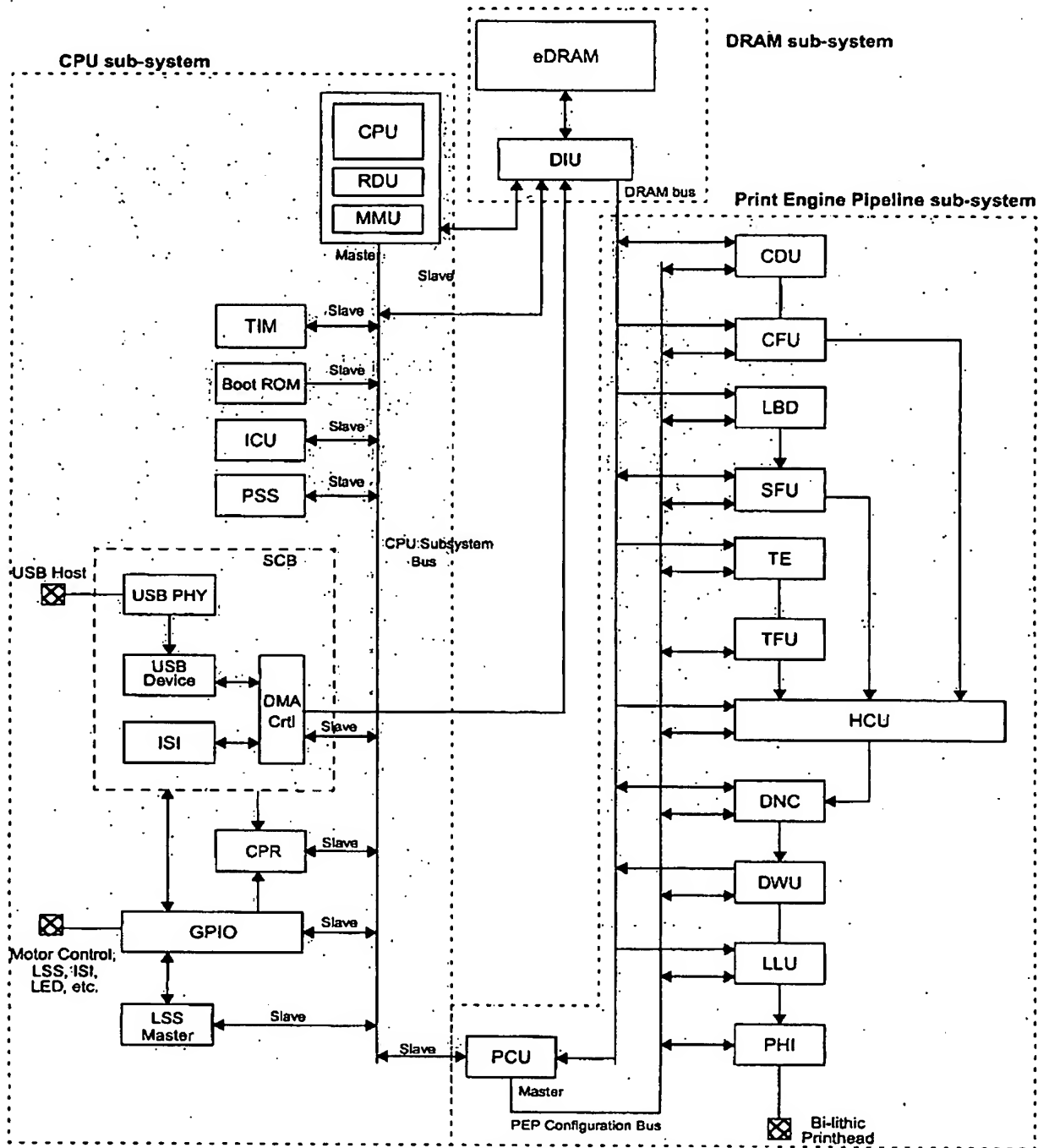
Figure 26. SoPEC toplevel block diagram

12.2   DEFINITIONS OF I/Os

Table 28. Serial Communications Block I/O

Portname	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	System reset signal. Active low.
pclk	1	In	System clock.
usb_clk	1	In	Clock for the USB controller block.
isi_cpr_reset_n	1	Out	Signal from the ISI indicating that ISI activity has been detected while in sleep mode and so the chip should be reset. Active low.
usb_cpr_reset_n	1	Out	Signal from the USB controller that a USB reset has occurred. Active low.
CPU Interface			
cpu_adr[n:2]	n-1	In	CPU address bus. Exact width is currently TBD as it is dependent on the address maps of imported IP
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
scb_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_fc[2:0]	3	In	CPU Function Code signals.
cpu_scb_sel	1	In	Block select from the CPU. When *cpu_scb_sel* is high both *cpu_adr* and *cpu_dataout* are valid
scb_cpu_rdy	1	Out	Ready signal to the CPU. When *scb_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the SCB and for a read cycle this means the data on *scb_cpu_data* is valid.
scb_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
scb_cpu_debug_valid	1	Out	Signal indicating that the data currently on *scb_cpu_data* is valid debug data
Interrupt signals			
dma_icu_irq	1	Out	DMA interrupt signal to the interrupt controller block.
isi_icu_irq	1	Out	ISI interrupt signal to the interrupt controller block.
usb_icu_irq	1	Out	USB interrupt signal to the interrupt controller block.
DIU interface			
scb_diu_wadr[21:5]	17	Out	Write address bus to the DIU
scb_diu_data[63:0]	64	Out	Data bus to the DIU.
scb_diu_wreq	1	Out	Write request to the DIU
diu_scb_wack	1	In	Acknowledge from the DIU that the write request was accepted.
scb_diu_wvalid	1	Out	Signal from the SCB to the DIU indicating that the data currently on the scb_diu_data[63:0] bus is valid
GPIO Interface			
isi_gpio_dout[1:0]	2	Out	ISI output data to GPIO pins
isi_gpio_e[1:0]	2	Out	ISI output enable to GPIO pins
gpio_isi_din[1:0]	2	In	Input data from GPIO pins to ISI

12.3 MULTI-SoPEC SYSTEMS

While single SoPEC systems are expected to form the majority of SoPEC systems the SoPEC device must also support its use in multi-SoPEC systems such as that shown in Figure 27 below. A SoPEC may be assigned any one of a number of identities in a multi-SoPEC system. A SoPEC may be one or more of a PrintMaster, a LineSyncMaster, an ISIMaster, a StorageSoPEC or an ISISlave SoPEC
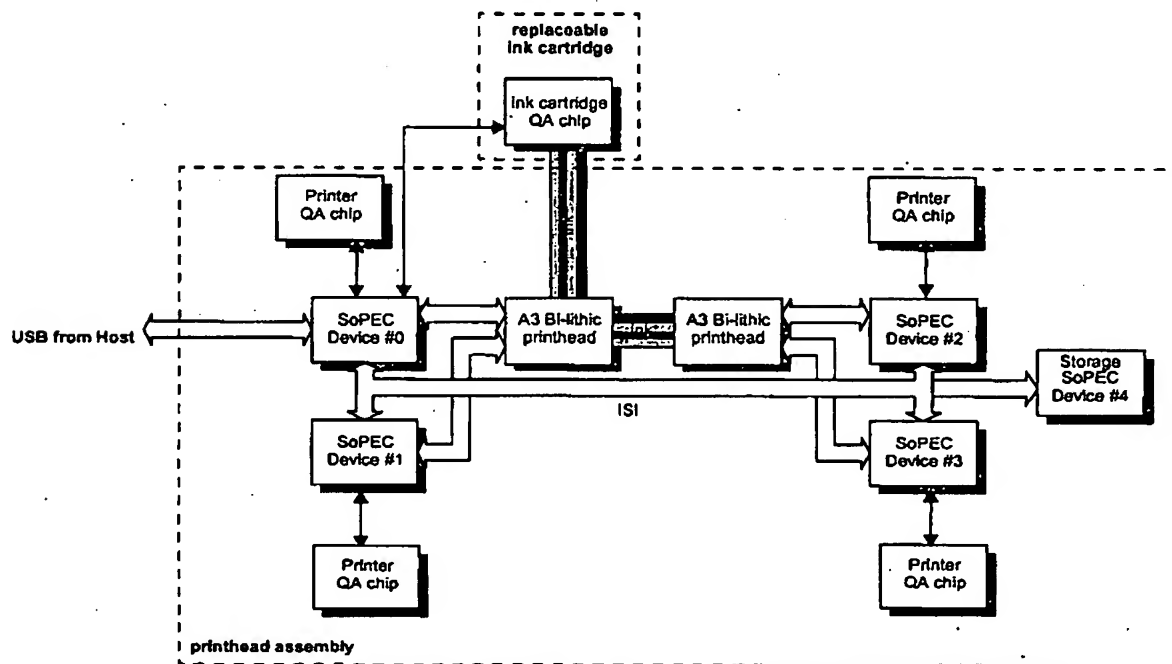
Figure 27. A3 duplex system featuring four printing SoPECs with a single SoPEC DRAM device

12.3.1 ISIMaster device

The ISIMaster is the only device allowed to drive the common ISI line (see Figure 28) and interfaces directly with the host. In most systems the ISIMaster will simply be the SoPEC connected to the USB bus. Future systems, however, may employ an ISI-Bridge chip to interface between the host and the ISI bus and in such systems the ISI-Bridge chip will be the ISIMaster. There can only be one ISIMaster on an ISI bus.

12.3.2 PrintMaster device

The PrintMaster device is responsible for co-ordinating all aspects of the print operation. This includes starting the print operation in all printing SoPECs and communicating status back to the host. When the ISIMaster is a SoPEC device it is also likely to be the PrintMaster as well. There may only be one Print-Master in a system and it is most likely to be a SoPEC device.

12.3.3 LineSyncMaster device

The LineSyncMaster device generates the *lsync* pulse that all SoPECs in the system must synchronize their line outputs with. Any SoPEC in the system could act as a LineSyncMaster although the PrintMaster is probably the most likely candidate. It is possible that the LineSyncMaster may not be a SoPEC device at all - it could, for example, come from some OEM motor control circuitry. There may only be one LineSyncMaster in a system.

12.3.4 Storage device

For certain printer types it may be realistic to use one SoPEC as a storage device without using its print engine capability - that is to effectively use it as an ISI-attached DRAM. A storage SoPEC would receive data from the ISIMaster (most likely to be an ISI-Bridge chip) and then distribute it to the other SoPECs as required. No other type of data flow (e.g. ISISlave -> storage SoPEC -> ISISlave) would need to be supported in such a scenario. The SCB supports this functionality at no additional cost because the CPU handles the task of transferring outbound data from the embedded DRAM to the ISI transmit buffer. The CPU in a storage SoPEC will have almost nothing else to do.

12.3.5 ISISlave device

Multi-SoPEC systems will contain one or more ISISlave SoPECs. An ISISlave SoPEC is primarily used to generate dot data for the printhead IC it is driving.

12.3.6 ISI-Bridge device

SoPEC is targeted at the low-cost small office / home office (SoHo) market. It may also be used in future systems that target different market segments which are likely to have a high speed interface capability. A future device, known as an ISI-Bridge chip, is envisaged which will feature both a high speed interface (such as USB2.0, Ethernet or IEEE1394) and one or more ISI interfaces. The use of multiple ISI buses would allow the construction of independent print systems within the one printer. The ISI-Bridge would be the ISIMaster for each of the ISI buses it interfaces to.

12.3.7 Host device

The host device will invariably be, but is not required to be, a PC. Any device that can act as a USB host or that can interface to an ISI-Bridge chip could be the host device. In particular, with the development of USB On-The-Go (USB OTG), it is possible that a number of USB OTG enabled products such as PDAs or digital cameras will be able to directly interface with a SoPEC printer.

12.4 TYPES OF COMMUNICATION

12.4.1 Communications with host

The host communicates directly with the ISIMaster in order to print pages. When the ISIMaster is a SoPEC, the communications channel is USB1.1.

12.4.1.1 Host to ISIMaster communication

The host will need to communicate the following information to the ISIMaster device:
- Communications channel configuration and maintenance information
- All data destined for PrintMaster, ISISlave or storage SoPEC devices. This data is simply relayed by the ISIMaster
- Mapping of virtual communications channels, such as USB endpoints, to ISI destination

12.4.1.2 ISIMaster to host communication

The ISIMaster will need to communicate the following information to the host:
- Communications channel configuration and maintenance information
- All data originating from the PrintMaster, ISISlave or storage SoPEC devices and destined for the host. · This data is simply relayed by the ISIMaster

12.4.1.3 Host to PrintMaster communication

The host will need to communicate the following information to the PrintMaster device:
- Program code for the PrintMaster
- Compressed page data for the PrintMaster
- Control messages to the PrintMaster
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.4.1.4 PrintMaster to host communication

The PrintMaster will need to communicate the following information to the host:
- Printer status information (i.e. authentication results, paper empty/jammed etc.)
- Dead nozzle information
- Memory buffer status information
- Power management status
- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.4.1.5 Host to ISISlave communication

All communication between the host and ISISlave SoPEC devices must take place via the ISIMaster. In the case of a SoPEC ISIMaster it is possible to configure each individual USB endpoint to act as a control channel to an ISISlave SoPEC if desired, although the endpoints will be more usually used to transport data. The host will need to communicate the following information to ISISlave devices over the comms/ISI:
- Program code for ISISlave SoPEC devices
- Compressed page data for ISISlave SoPEC devices
- Control messages to the ISISlave SoPEC (where a control channel is supported)
- Tables and static data required for printing e.g. dead nozzle tables, dither matrices etc.
- Authenticatable messages to upgrade the printer's capabilities

12.4.1.6 ISISlave to host communication

All communication between the ISISlave SoPEC devices and the host must take place via the ISIMaster. The ISISlave will need to communicate the following information to the host over the comms/ISI:
- Responses to the host's control messages (where a control channel is supported)
- Dead nozzle information from the ISISlave SoPEC.
- Encrypted SoPEC_id for use in the generation of PRINTER_QA keys during factory programming

12.4.2   Communication over ISI

12.4.2.1 ISIMaster to PrintMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:
- All data from the host destined for the PrintMaster (see section 12.4.1.3). This data is simply relayed by the ISIMaster

12.4.2.2 PrintMaster to ISIMaster communication

The ISIMaster and PrintMaster will often be the same physical device. When they are different devices then the following information needs to be exchanged over the ISI:
- All data from the PrintMaster destined for the host (see section 12.4.1.4). This data is simply relayed by the ISIMaster

12.4.2.3 ISIMaster to ISISlave communication

The ISIMaster may wish to communicate the following information to the ISISlaves:
- All data (including program code such as ISIId enumeration) originating from the host and destined for the ISISlave (see section 12.4.1.5). This data is simply relayed by the ISIMaster
- wake up from sleep mode

12.4.2.4 ISISlave to ISIMaster communication

The ISISlave may wish to communicate the following information to the ISIMaster:
- All data originating from the ISISlave and destined for the host (see section 12.4.1.6). This data is simply relayed by the ISIMaster

12.4.2.5 PrintMaster to ISISlave communication

When the PrintMaster is not the ISIMaster all ISI communication is done in response to ISI ping packets (see 12.6.4.5).   When the PrintMaster is the ISIMaster then it will of course communicate directly with the ISISlaves. The PrintMaster SoPEC may wish to communicate the following information to the ISISlaves:
- Ink status e.g. requests for *dotCount* data i.e. the number of dots in each color fired by the printheads connected to the ISISlaves
- configuration of GPIO ports e.g. for clutch control and lid open detect
- power down command telling the ISISlave to enter sleep mode
- ink cartridge fail information

This list is not complete and the time constraints associated with these requirements have yet to be determined.

In general the PrintMaster may need to be able to:
- send messages to an ISISlave which will cause the ISISlave to return the contents of ISISlave registers to the PrintMaster or
- to program ISISlave registers with values sent by the PrintMaster

This should be under the control of software running on the CPU which writes messages to the ISI/SCB interface.

12.4.2.6 ISISlave to PrintMaster communication

ISISlaves may need to communicate the following information to the PrintMaster:
- ink status e.g. *dotCount* data i.e. the number of dots in each color fired by the printheads connected to the ISISlaves
- band related information e.g. finished band interrupts
- page related information i.e.buffer underrun, page finished interrupts
- MMU security violation interrupts
- GPIO interrupts and status e.g. clutch control and lid open detect
- printhead temperature
- printhead dead nozzle information from SoPEC printhead nozzle tests
- power management status

This list is not complete and the time constraints associated with these requirements have yet to be determined.

As the ISI is an insecure interface commands issued over the ISI should be of limited capability e.g. only limited register writes allowed. The software protocol needs to be constructed with this in mind. In general ISISlaves may need to return register or status messages to the PrintMaster or ISIMaster. They may also need to indicate to the PrintMaster or ISIMaster that a particular interrupt has occurred on the ISISlave. This should be under the control of software running on the CPU which writes messages to the ISI block.

12.4.2.7 ISISlave to ISISlave communication

It is currently not anticipated that there will be any *direct* communication between ISISlave SoPECs. However they can communicate *indirectly* via the ISIMaster SoPEC. The most likely scenario for such a communication mechanism when the PrintMaster is not the ISIMaster (see sections 12.4.2.5 and 12.4.2.6 for a description of the information exchanged between a PrintMaster and an ISISlave). ISISlave to ISISlave communication would also be required when sending data stored in a storage SoPEC device to an ISISlave.

12.5   USB

The USB1.1 interface for the printer should consist of the USB connector, the necessary discretes for USB signalling and the SoPEC device. A SoPEC printer will act as a self-powered, full-speed device and SoPEC itself will not draw any power from the USB cable. It will support control and bulk transfers. Interrupt transfers are not considered necessary because the required interrupt-type functionality can be achieved by sending query messages over the control channel on a scheduled basis. There is no requirement to support either isochronous or low-speed transfers. The USB controller must support at least 5 USB endpoints: a control endpoint (endpoint 0) and 4 bulk-data type endpoints. These 4 bulk-data type endpoints can be used for the transfer of any type of data: compressed page data, program data or control messages. They may also be mapped on to any target destination in a multi-SoPEC system i.e. configuration is completely programmable. They are envisaged as always being used as USB IN endpoints i.e. they will transport data from the host to SoPEC. Any feedback data (e.g. status information) will be returned to the host on the control channel (endpoint 0).

The USB device enumeration process will be handled by the SoPEC CPU and USB controller. Note that this requires the on-chip ROM to contain all the required USB driver code. This is not expected to be the full USB driver but rather a "USB-lite" driver that has sufficient functionality to download a program to DRAM.

Details of the configuration registers and interface signals will be provided when the implementation IP for the USB controller core has been selected. There are several potential candidates for the USB1.1 con-

troller that are being evaluated in terms of cost, maturity, licensing requirements/restrictions, quality of deliverables etc. - as already mentioned the choice of silicon vendor is likely to play a large part in selecting the USB controller.

12.5.1 ISIMaster/ISISlave Identification

While the USB controller is used for data transfer if a SoPEC is an ISIMaster it may, in certain cases, also be used to transfer data to an ISISlave. If the USB is not used for data transfer the device will certainly be an ISISlave. In this case the USB pins could be used to identify the device as an ISISlave as the USB device controller is expected to allow the single-ended quiescent state of the USB pins to be read by the CPU either directly or indirectly (as there should be a register indicating whether the USB controller is operating as a full-speed or low-speed device). We adopt the convention that an ISIMaster SoPEC has its USB pins configured for full-speed operation (i.e. a pull-up resistor on D+) and an ISISlave SoPEC has its USB pins configured for low-speed operation (i.e. a pull-up resistor on D-). This allows the ROM bootcode to quickly determine whether the SoPEC is an ISIMaster or ISISlave without needing to wait for USB activity. While the ISISlave SoPEC's USB controller believes it is a low-speed device it is never used and may be disabled completely (if possible) once the device has been identified as an ISISlave. Note that other combinations on the D+ and D- lines may result in unreliable operation of the USB controller.

The SoPECs identity as an ISIMaster or ISISlave may also be determined from USB or ISI activity. If activity is seen on USB endpoints 2-4 then the device is an ISIMaster (note that it is not neccessarily an ISIMaster if activity is only seen on endpoints 0 or 1) and the ISI may automatically configure itself as an ISIMaster in this situation. If the ISI receives ping packets then it is an ISISlave as only the ISIMaster can send ping packets.

The most suitable ISIMaster/ISISlave identification scheme (i.e. use of USB pins or looking for USB/ISI activity) can be chosen by the software for any given printer.

12.5.2 Wake-up from sleep mode

The SoPEC will be placed in sleep mode after a suspend command is received by the USB controller. The extent of power-down in sleep mode is currently TBD (different silicon vendors offer different options) but it is expected to involve the loss of DRAM contents at a minimum. The USB controller (or portions of it) will continue to be powered and clocked in sleep mode. It is likely that a USB reset, as opposed to a device resume, will be required to bring SoPEC out of its sleep state as the sleep state is hoped to be logically equivalent to the power down state. The exact reawakening mechanism will be finalised when the sleep state is more precisely defined and the particular implementation of the USB controller is chosen.

The USB reset signal originating from the USB controller will be propagated to the CPR (as *usb_cpr_reset_n*) if the *USBWakeupEnable* bit of the *WakeupEnable* register (see Table 38) has been set. The *USBWakeupEnable* bit should therefore be set just prior to entering sleep mode.

There are no conditions that require the SoPEC to initiate a USB device wake-up (i.e. where SoPEC signals resume to the host after being suspended by the host).

12.5.3 USB Speed

The USB speed will be determined by amount of activity from other devices that share the USB bus with the printer and the responsiveness of the host in handling USB interrupts. To guarantee bandwidth to the printer it is recommended that no other devices are active on the USB bus between the printer and the host. If the printer is connected to a USB2.0 host or hub it may limit the bandwidth available to other devices connected to the same hub but it would not significantly affect the bandwidth available to other devices upstream of the hub. Used in the recommended configuration it is expected that an effective bandwidth of 8-9 Mbit/s will be achieved.

12.6   ISI (INTER SoPEC INTERFACE)

The ISI is utilised in all system configurations requiring more than one SoPEC. An example of such a system which requires four SoPECs for duplex A3 printing and an additional SoPEC used as a storage device is shown in Figure 27.

The ISI performs much the same function between an ISISlave SoPEC and the ISIMaster as the USB connection performs between the ISIMaster and the host. This includes the transfer of all program data, compressed page data and message (i.e. commands or status information) passing between the ISIMaster and the ISISlave SoPECs. Existing requirements indicate that it is sufficient for the ISIMaster to initiate all communication with the ISISlaves.

12.6.1   ISIMaster/ISISlave identification and ISISlave enumeration

Section 12.5.1 details how a SoPEC is configured as an ISIMaster or ISISlave. The ISIId is established by software downloaded over the ISI (in broadcast mode) which looks at the input levels on a number of GPIO pins to determine the ISIId. For any given printer that uses a multi-SoPEC configuration it is expected that there will always be enough free GPIO pins on the ISISlaves to support this enumeration mechanism.

12.6.2   Wake-up from sleep mode

Either the PrintMaster SoPEC or the host may place any of the ISISlave SoPECs in sleep mode prior to going into sleep mode itself. The ISISlave device should then ensure that its *ISIWakeupEnable* bit of the *WakeupEnable* register (see Table 38) is set prior to entering sleep mode. In an ISISlave device the ISI block will continue to receive power and clock during sleep mode so that it may monitor the *gpio_isi_din* lines for activity. When ISI activity is detected during sleep mode and the *ISIWakeupEnable* bit is set the ISI asserts the *isi_cpr_reset_n* signal. This will bring the rest of the chip out of sleep mode by means of a wakeup reset. See chapter 16 for more details of reset propagation.

12.6.3   ISI speed

The ISI will need to run at speed that will allow error free transmission on the PCB while minimising the buffering and hardware requirements on SoPEC. While an ISI speed of 10 Mbit/s is adequate to match the effective USB1.1 bandwidth it would limit the system performance when a high-speed connection (e.g. USB2.0, IEEE1394) is used to attach the printer to the PC. Although they would require the use of an extra ISI-Bridge chip such systems are envisaged for more expensive printers (compared to the low-cost basic SoPEC powered printers that are initially being targeted) in the future.

An ISI line speed (i.e. the speed of each individual ISI wire) of 32 Mbit/s is therefore proposed as it will allow ISI data to be oversampled 5 times (at a *pclk* frequency of 160MHz). The total bandwidth of the ISI will depend on the number of pins used to implement the interface. The current expectation is that two pins will be used, giving a peak raw bandwidth of 64 Mbit/s, and this is the scenario that is used in this document. However the ISI protocol will work equally well if four pins are used for transmission/reception and this would give a peak raw bandwidth of 128 Mbit/s. The number of pins available for the ISI is currently under investigation as part of the package selection process. With either a two or four pin ISI solution a 32 Mbit/s line speed would allow the movement of data in to and out of a storage SoPEC (as described in 12.3.4 above), which is the most bandwidth hungry ISI use, in a timely fashion.

The maximum effective bandwidth of a two wire ISI, after allowing for protocol overheads and bus turnaround times, is expected to be approx. 50 Mbit/s.

12.6.4 ISI protocol

The ISI is a serial interface utilizing a two wire half-duplex configuration as shown in Figure 28 below. An ISIMaster must always be present and up to 14 ISISlaves may also be on the ISI bus. The ISI bus enables broadcasting of data, ISIMaster to ISISlave communication, ISISlave to ISIMaster communication and ISISlave to ISISlave communication. Flow control, error detection and retransmission of errored packets is also supported. ISI transmission is asynchronous and a *Start* field is present in every transmitted packet to ensure synchronization for the duration of the packet. Bit-stuffing is required as it is expected that synchronization cannot be guaranteed for the length of the longest allowed packet.[1] **Open Issue:** This should be confirmed with the spec of the crystal used with SoPEC. We may wish to constrain the spec of *xtalin* and also *xtalin* for the ISI-Bridge chip to ensure the ISI cannot drift out of sync during packet reception.
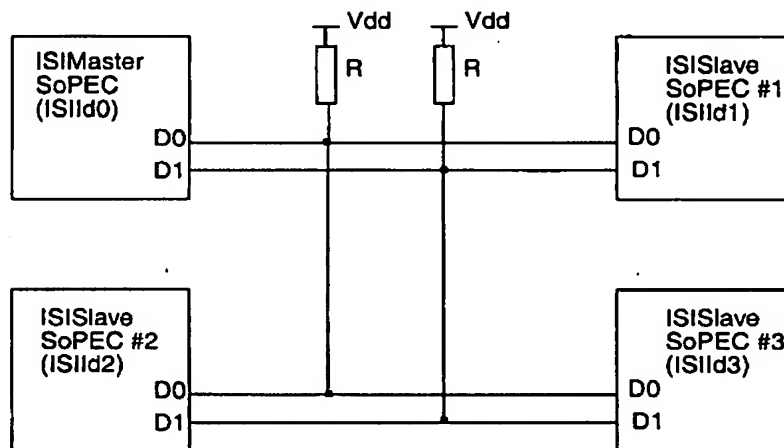
Figure 28. ISI configuration with four SoPEC devices

To maximize the effective ISI bandwidth while minimising pin requirements a two wire half-duplex inter-leaved transmission scheme is used. Figure 29 below shows how a 16-bit word is transmitted from an ISI-Master to an ISISlave. Data is interleaved on a bit-by-bit basis over the two ISI lines and this requires all ISI packets to be an even number of bits in length. This interleaving could easily be extended to four pins if required.

All ISI transactions are initiated by the ISIMaster and every non-broadcast data packet needs to be acknowledged by the addressed recipient. An ISISlave may only transmit when it receives a ping packet (see section 12.6.4.5) addressed to it. To avoid bus contention all ISI devices must wait one bit-time (5 *pclk* cycles) after detecting the end of a packet before transmitting a packet (assuming they are required to transmit). All non-transmitting ISI devices must tristate their Tx drivers to avoid line contention. A pull-up resistor is therefore required on both ISI lines to reduce the possibility of false data detection. The ISI pro-tocol is defined to avoid devices driving out of order (e.g. when an ISISlave is no longer being addressed). As the ISI will use standard I/O pads there will be no physical collision detection mechanism.

---

1. Current max packet size ~= 290 bits = 145 bits per line (on a 2 wire ISI) = 725 160MHz cycles. Thus the *pclk*s in the two communicat-ing ISI devices should not drift by more than one cycle in 725 i.e. 1379 ppm. Careful analysis of the crystal, PLL and oscillator specs and the sync detection circuit is needed here to ensure our solution is robust.
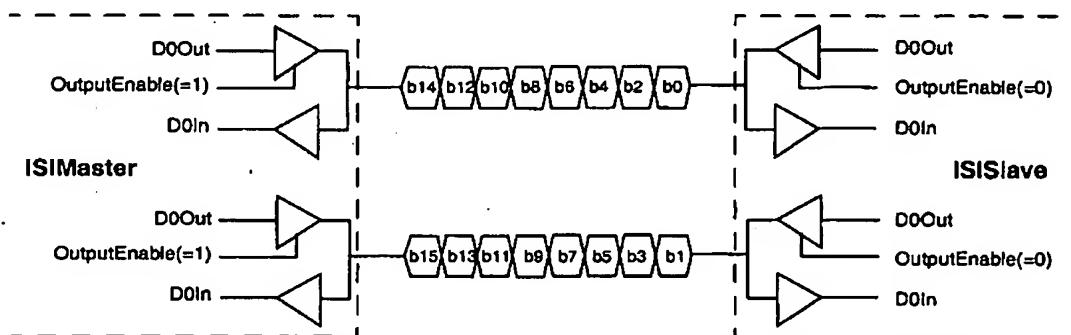
Figure 29. Half-duplex interleaved transmission from ISIMaster to ISISlave

There are three types of ISI packet: a long packet (used for data transmission), a ping packet (used by the ISIMaster to prompt ISISlaves for packets) and a short packet (used to acknowledge receipt of a packet). All ISI packets are delineated by a *Start* and *Stop* fields and transmission is atomic i.e. an ISI packet may not be split or halted once transmission has started.

12.6.4.1 ISI transactions

The different types of ISI transactions are outlined in Figure 30 below. As described later all NAKs are inferred and ACKs are not addressed to any particular ISI device.
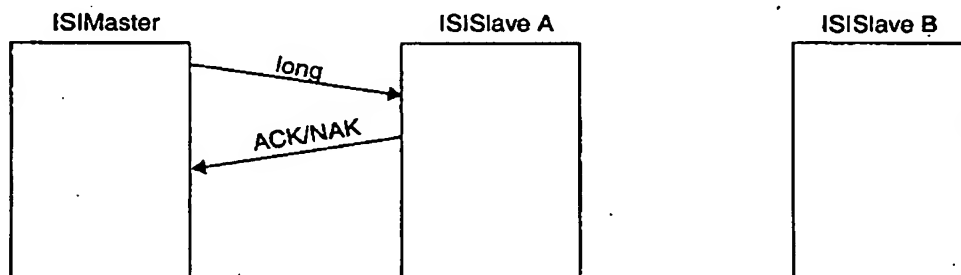
Transaction 1: Long packet to an addressed ISISlave
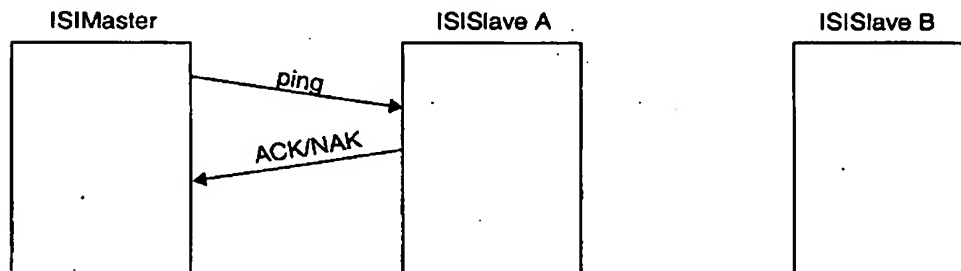
Transaction 2: Ping packet to an addressed ISISlave. ISISlave has nothing to send
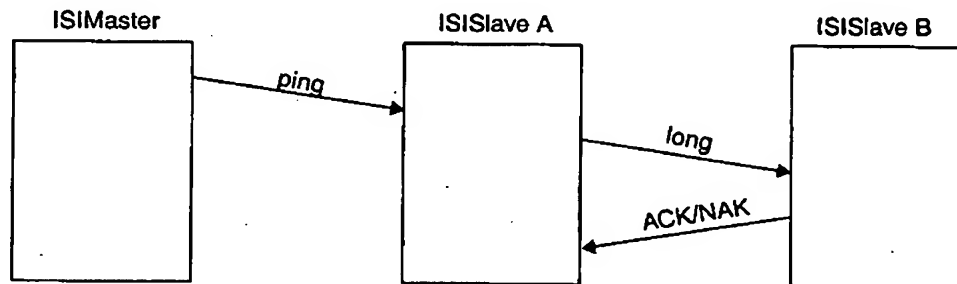
Transaction 3: Ping packet to an addressed ISISlave. ISISlaveA responds with a long packet to ISISlaveB and ISISlaveB responds with an ACK or NAK.
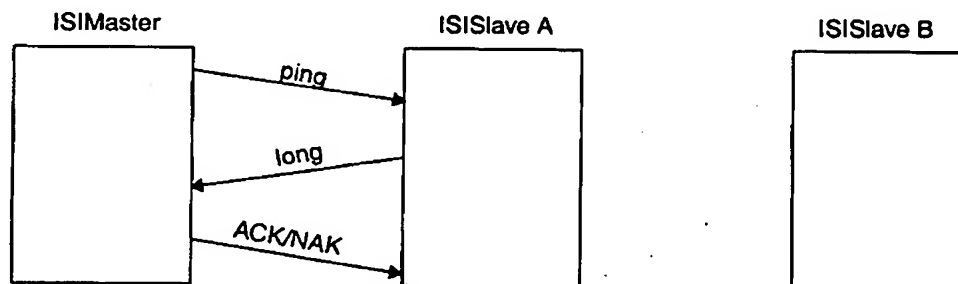
Transaction 4: Ping packet to an addressed ISISlave. ISISlaveA responds with a long packet to the ISIMaster and the ISIMaster responds with an ACK or NAK.

Figure 30. ISI transactions

12.6.4.2 Start field description and bit stuffing

The *Start* field serves two purposes: To allow the start of a packet be unambiguously identified and to allow the receiving device synchronise to the data stream. The symbol, or data value, used to identify a *Start* field must not legitimately occur in the ensuing packet. Bit stuffing is used to guarantee that the Start symbol will be unique in any valid (i.e. error free) packet. The *Start* symbol should therefore be sufficiently long to ensure that the bit stuffing overhead is low but should still be short enough to reduce its own contribution to the packet overhead. A *Start* bit length of 8 bits is therefore used as it is an effective compromise between these two constraints. The *Start* field, like every byte in a packet, is transmitted with its rightmost (lsb) bit first.

If the correct symbol value is used bit stuffing offers the further advantage of forcing transitions on the ISI lines which will allow synchronization be maintained. Unfortunately a symbol value that is good for forcing transitions (e.g. 0x00) is not good for guaranteeing initial synchronization and vice versa i.e. a symbol such as 0xAA would ensure initial synchronization but cannot prevent synchronization being lost if a long run of zeroes or ones is subsequently transmitted.

To resolve this conflict the *Start* symbol will be 0xAA and three different types of bit stuffing are used. Whenever 0xAA is encountered in the data stream a 0 is inserted before the msb resulting in the 9-bit value 0x12A (i.e. b10101010 -> b100101010). To ensure transitions occur during a long run of zeroes a 1 is inserted after 7 zeroes thus 0x00 becomes 0x080 (i.e. b00000000 -> b010000000). Likewise to ensure

transitions will occur during a run of ones a 0 is inserted after 7 ones and so 0xFF becomes 0x17F (i.e. b11111111 -> b101111111). The receiving ISI device must detect these special values and strip out the inserted ones and zeroes.

Note that any violation of bit stuffing will result in the *FrameError* status bit being set and the incoming packet will be treated as an errored packet. Furthermore if the *Start* field is not received as 0xAA the *FrameError* status bit is set and incoming data is discarded until a correct *Start* field is detected.

In a truly random data such a bit stuffing scheme could cause an overhead of approx. 0.15%. While the data transmitted over the ISI will not be truly random (0x00 and 0xFF are likely to occur more often than they would in a random data set) the overhead should remain low and will never exceed 11.1% (i.e. 1 in every 9 bits).

12.6.4.3 Stop field description

A 2-bit *Stop* field (= b11) is used to ensure that both lines return to the high state before the next packet is transmitted. Two bits are required because the *Stop* field will be interleaved over both ISI lines (4 bits would be used in a 4 wire ISI). The *Stop* field is not subject to bit stuffing because bit stuffing could result in the final transmitted bit being a 0 on one of the ISI lines.

12.6.4.4 ISI long packet description

The format of a long ISI packet is shown in Figure 31 below. Data may only be transferred between ISI devices using a long packet as both the short and ping packets have no payload field. Except in the case of a broadcast packet, the receiving ISI device will always reply to a long packet with either an explicit ACK (no error detected in received packet) or an inferred NAK (an error was detected in the received packet).
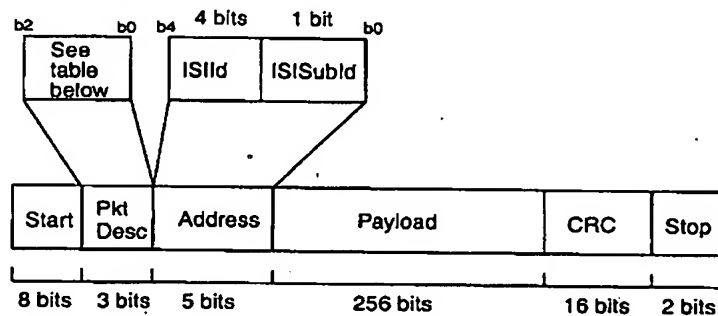
Figure 31. ISI long packet

All long packets begin with the *Start* field as described earlier. The *PktDesc* field is described in Table 29.

Table 29. PktDesc field description

Bit	Description
0	Packet type indicator: 1 - Short packet 0 - Non-short (i.e. long/ping) packet

Table 29. PktDesc field description

Bit	Description
1	Data payload present indicator 1 - This packet contains payload (i.e. long packet) 0 - This packet has no payload
2	Sequence bit value. Only valid for long packets. See section 12.6.4.8 for a description of sequence bit operation

Any ISI device in the system may transmit a long packet but only the ISIMaster may initiate an ISI trans-action using a long packet. An ISISlave may only send a long packet in reply to a ping message from the ISIMaster. A long packet from an ISISlave may be addressed to any ISI device in the system although the ISIMaster (or the PrintMaster if it is a different device) will be the usual recipient.

The *Address* field is straightforward and complies with the ISI naming convention described in section 12.7.

The payload field is exactly what is in the transmit buffer of the transmitting ISI device and gets copied into the receive buffer of the addressed ISI device(s). When present the payload field is always 256 bits.

To ensure strong error detection a 16-bit CRC is appended. This CRC is calculated over the entire packet (excluding the *Start* and *Stop* fields). The HDLC standard CRC-16 (i.e. $G(x) = x^{16} + x^{12} + x^5 + 1$) is to be used for this calculation, which is to be performed serially.

12.6.4.5 ISI ping packet

The ISI ping packet is used to allow ISISlaves transmit on the ISI bus. As can be seen from Figure 32 below the ping packet is cab be viewed as a special case of the long packet. In other words it is a long packet without any payload, whose *PktDesc* field is always b000 and whose ISISubId is always 1. The ISISubId is unused in ping packets because the ISIMaster is addressing the ISI device rather than one of the DMA channels in the device. The ISISlave may address any ISIId.ISISubId in response if it wishes. The ISISlave will respond to a ping packet with either an explicit ACK (if it has nothing to send), an inferred NAK (if it detected an error in the ping packet) or a long packet (containing the data it wishes to send). Note that inferred NAKs do not result in the retransmission of a ping packet. This is because the ping packet will be retransmitted on a predetermined schedule (see 12.6.4.10 for more details).

Figure 32. ISI ping packet

An ISISlave should never respond to a ping message to the broadcast ISIId as this must have been sent in error. An ISI ping packet will never be sent in response to any packet and may only originate from an ISI-Master.

12.6.4.6 ISI short packet description

The ISI short packet is only 14 bits long, including the *Start* and *Stop* fields. A value of b1001 is proposed for the ACK symbol. As a 16-bit CRC is inappropriate for such a short packet it is not used. In fact there is only one valid value for a 14-bit short ACK packet as the *Start*, AC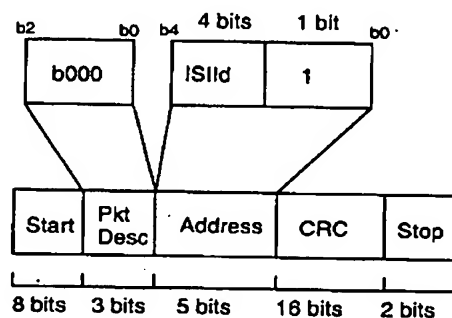K and *Stop* symbols all have fixed values. Short packets are only used for acknowledgements (i.e explicit ACKs). The format of a short ISI packet is shown in Figure 33 below.

Start	Ack Symbol	Stop
8 bits	4 bits	2 bits
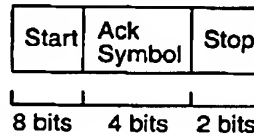
Figure 33. Short ISI packet

12.6.4.7 Error detection and retransmission

The 16-bit CRC will provide a high degree of error detection and the probability of transmission errors occurring is very low as the transmission channel (i.e. PCB traces) will have a low inherent bit error rate. The number of undetected errors should therefore be minute. A simple retransmission mechanism frees the CPU from getting involved in error recovery for most errors because the probability of a transmission error occurring more than once in succession is very, very low in normal circumstances.

After each non-short ISI packet is transmitted the transmitting device will open a reply window. The size of the reply window will be 9 bit times (i.e. 14 bits transmitted on two wires plus 2 bit times to allow for bus turnarounds and timing differences) when a short packet is expected and 147 bit times (i.e. 290 bits transmitted on two wires plus 2 bit times to allow for bus turnarounds and timing differences) when a long packet is expected in reply.

When a packet has been received without any errors the receiving ISI device must transmit its acknowledge packet (which may be either a long or short packet) before the reply window closes. When detected errors do occur the receiving ISI device will not send any response. The transmitting ISI device interprets this lack of response as a NAK indicating that errors were detected in the transmitted packet or that the receiving device was unable to receive the packet for some reason. If a long packet was transmitted the transmitting ISI device will keep the transmitted packet in its transmit buffer for retransmission. If the transmitting device is the ISIMaster it will retransmit the packet immediately while if the transmitting device is an ISISlave it will retransmit the packet in response to the next ping it receives from the ISIMaster.

The transmitting ISI device will continue retransmitting the packet when it receives a NAK until it either receives an ACK or the number of retransmission attempts equals the value of the *NumRetries* register. If the transmission was unsuccessful then the transmitting device sets the *TxError* bit in its *ISIStatus* register. The receiving device also sets the *RxError* bit in its *ISIStatus* register whenever it detects *NumRetries* + 1 errored packets in succession. The *NumRetries* registers in all ISI devices should therefore be set to the same value for consistent operation. Note that successful transmission or reception of ping packets do not affect retransmission operation. **Open Issue:** In the case of an ISI device receiving a packet in error from an ISISlave the *NumRetries* count will be reset if it subsequently receives an error free packet from any ISI device (which may not be the ISISlave that transmitted the errored packet). Thus the *RxError* operation is only effective for ISIMaster to ISISlave transactions as these are the only ones where retransmissions will be sequential. Either we live with this or we could implement a NumRetriesCount window which would allow all NAKs within a specified window to be counted. If *NumRetries* is exceeded within this window then we have a *RxError* otherwise we can reset the count.

Note that either a transmit or receive error will cause the ISI to stop transmitting or receiving respectively. CPU intervention will be required to resolve the source of the problem and to restart the ISI transmit or receive operation. Transmit or receive errors should be extremely rare and their occurrence will most likely indicate a serious problem.

Note that broadcast packets are never acknowledged to avoid contention on the common ISI lines. If an ISISlave detects an error in a broadcast packet it must use the message passing mechanism described earlier to alert the ISIMaster to the error.

12.6.4.8 Sequence bit operation

To ensure that communication between transmitting and receiving ISI devices is correctly ordered a sequence bit is included in every long packet to keep both devices in step with each other. Sequence bits are not used for short or ping packets as they are not used for data transmission. In addition to the transmitted sequence bit all ISI devices keep two local sequence bits, one for each ISISubId. Furthermore each ISI device maintains a transmit sequence bit for each ISIId and ISISubId it is in communication with. For packets sourced from the host (via USB) the transmit sequence bit is contained in the relevant *USBEPnDest* register while for packets sourced from the CPU the transmit sequence bit is contained in the *CPUISITxBuffCntrl* register. The sequence bits for received packets are stored in *DMA0SeqBit* and *DMA1SeqBit* registers. All ISI devices will initialise their sequence bits to 0 after reset. It is the responsibility of software to ensure that the sequence bits of the transmitting and receiving ISI devices are correctly initialised each time a new source is selected for any ISIId.ISISubId channel.

Sequence bits are not used in all broadcast and ping packets. Each SoPEC may also ignore the sequence bit on either of its ISISubId channels by setting the appropriate bit in the *SequenceMask* register. The sequence bit should be ignored for ISISubId channels that will carry data that can originate from more than one source and is self ordering e.g. control messages.

A receiving ISI device will toggle its sequence bit addressed by the ISISubId only when the receiver is able to accept data and receives an error-free data packet addressed to it. The transmitting ISI device will toggle its sequence bit for that ISIId.ISISubId channel only when it receives a valid ACK handshake from the addressed ISI device.

Figure 34 shows the transmission of two long packets with the sequence bit in both the transmitting and receiving devices toggling from 0 to 1 and back to 0 again. The toggling operation will continue in this manner in every subsequent transmission until an error condition is encountered.
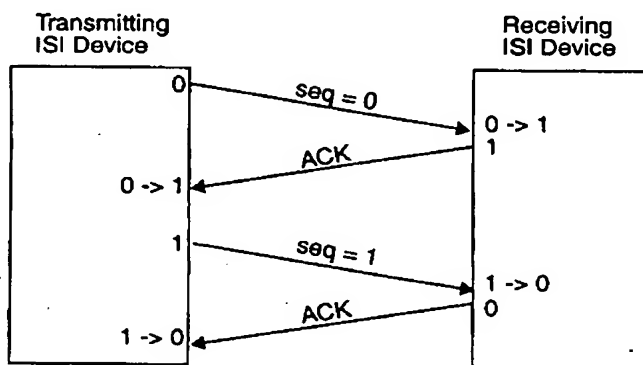
Figure 34. Successful transmission of two long packets with sequence bit toggling

When the receiving ISI device detects an error in the transmitted long packet or is unable to accept the packet (because of full buffers for example) it will not return any packet and it will not toggle its local sequence bit. An example of this is depicted in Figure 35. The absence of any response prompts the transmitting device to retransmit the original (seq=0) packet. This time the packet is received without any errors (or buffer space may have been freed) so the receiving ISI device toggles its local sequence bit and responds with an ACK. The transmitting device then toggles its local sequence bit to a 1 upon correct receipt of the ACK.
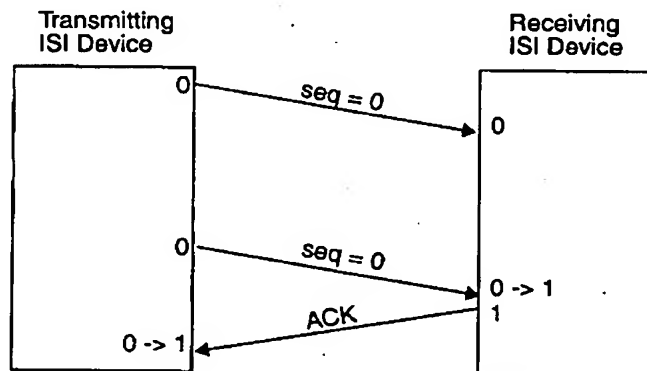
Figure 35. Sequence bit operation with errored long packet

However it is also possible for the ACK packet from the receiving ISI device to be corrupted and this scenario is shown in Figure 36. In this case the receiving device toggles its local sequence bit to 1 when then long packet is received without error and replies with an ACK to the transmitting device. The transmitting device detects an error in the ACK packet and so will not change its local sequence bit. It then retransmits the seq=0 long packet. When the receiving device finds that there is a mismatch between the transmitted sequence bit and the expected (local) sequence bit is discards the long packet and replies with an ACK. When the transmitting ISI device correctly receives the ACK it updates its local sequence bit to a 1, thus restoring synchronization. Note that when the *SequenceMask* bit for the addressed ISISubId is set then the retransmitted packet is not discarded and so a duplicate packet will be received. The data contained in the packet should be self-ordering and so the software handling these packets (most likely control messages) is expected to deal with this eventuality.
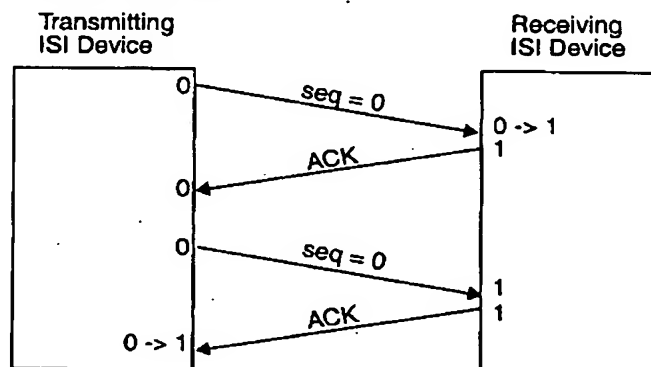
Figure 36. Sequence bit operation with ACK error

12.6.4.9 Flow control

The ISI also supports flow control by treating it in exactly the same manner as an error in the received packet. Because the SCB enjoys greater guaranteed bandwidth to DRAM than both the ISI and USB can supply flow control should not be required during normal operation. Any blockage on a DMA channel will soon result in the *NumRetries* value being exceeded and transmission to that DMA channel being halted. Because flow control is treated in the same manner as an error in the received packet neither the transmitting nor the receiving ISI device will be able to differentiate the cause of a *TxError* or *RxError*.

12.6.4.10 Auto-ping operation

While the CPU of the ISIMaster could send a ping packet by writing the appropriate header to the *CPUISITxBuffCntrl* register it is expected that all ping packets will be generated in the ISI itself. The use of automatically generated ping packets ensures that ISISlaves will be given access to the ISI bus with a programmable minimum guaranteed frequency in addition to whenever it is idle. Five registers facilitate the automatic generation of ping messages within the ISI: *PingSchedule0, PingSchedule1, PingSchedule2, ISITotalPeriod* and *ISILocalPeriod*. Auto-pinging can be enabled or disabled by writing to the *AutoPingEnable* bit of the *ISICntrl* register.

Each bit of the 14-bit *PingScheduleN* register corresponds to an ISIId that is used in the *Address* field of the ping packet and a 1 in the bit position indicates that a ping packet is to be generated for that ISIId. A 0 in any bit position will ensure that no ping packet is generated for that ISIId. As ISISlaves may differ in their bandwidth requirement (particularly if a storage SoPEC is present) three different PingSchedule registers are used to allow an ISISlave receive up to three times the number of pings as another active ISISlave. When the ISIMaster is not sending long packets (sourced from either the CPU or USB in the case of a SoPEC ISIMaster) ISI ping packets will be transmitted according to the pattern given by the three *PingScheduleN* registers. The ISI will start with the lsb of *PingSchedule0* register and work its way from lsb through msb of each of the *PingScheduleN* registers. When the msb of *PingSchedule2* is reached the ISI returns to the lsb of *PingSchedule0* and continues to cycle through each bit position of each *PingScheduleN* register.

With the addition of auto-ping operation we now have three potential sources of packets in an ISIMaster SoPEC: USB, CPU and auto-ping. Arbitration between the CPU and USB for access to the ISI is handled outside the ISI (see section 12.7.7) but arbitration between auto-ping packets and CPU/USB originating packets, which we will refer to as local packets, happens within the ISI. To ensure that local packets get priority whenever possible and that ping packets can have some guaranteed access to the ISI we use two 4-bit counters whose reload value is contained in the *ISITotalPeriod* and *ISILocalPeriod* registers. As we will see in 12.6.4.1 every ISI transaction is initiated by the ISIMaster transmitting either a long packet or a ping packet. The *ISITotalPeriod* counter is decremented for every ISI transaction when contention occurs (i.e. both a ping and a local packet wish to transmit) while the *ISILocalPeriod* counter is decremented for every local packet that is transmitted. Neither counter is decremented by a retransmitted packet.

The amount of guaranteed ISI bandwidth allocated to both local and ping packets is determined by the values of the *ISITotalPeriod* and *ISILocalPeriod* registers. Local packets will always be given priority when the *ISILocalPeriod* counter is non-zero. Ping packets will be given priority when the *ISILocalPeriod* counter is zero and the *ISITotalPeriod* counter is still non-zero. Both the *ISITotalPeriod* and *ISILocalPeriod* counters are reloaded by the next local packet transmit request after the *ISITotalPeriod* counter has reached zero. This reload policy minimises the maximum latency for ping packets at the expense of maximum latency for local packets.

Note that ping packets are quite likely to get more than their guaranteed bandwidth as they will be transmitted whenever the ISI bus is idle (i.e. no pending local packets) and so do not decrement either counter. Local packets on the other hand will never get more than their guaranteed bandwidth because each local packet transmitted decrements both counters. The difference between the values of the *ISITotalPeriod* and *ISILocalPeriod* registers determines the number of automatically generated ping packets that are guaran-

teed to be transmitted every *ISITotalPeriod* number of ISI transactions. If the *ISITotalPeriod* and *ISILocalPeriod* values are the same then the local packets will always get priority and could totally exclude ping packets if the CPU always has packets to send.

For example if *ISITotalPeriod* = 0xC; *ISILocalPeriod* = 0x8; *PingSchedule0* = 0x07; *PingSchedule1* = 0x06 and *PingSchedule2* = 0x04 then four ping messages are guaranteed to be sent in every 12 ISI transactions. Furthermore ISIId3 will receive 3 times the number of ping packets as ISId1 and ISIId2 will receive twice as many as ISId1. Thus over a period of 36 contended ISI transactions (allowing for two full rotations through the three *PingScheduleN* registers) when local packets are always pending 24 local packets will be sent, ISId1 will receive 2 ping packets, ISId2 will receive 4 pings and ISId3 will receive 6 ping packets. If local traffic is less frequent then the ping frequency will automatically adjust upwards to consume all idle ISI bandwidth.

12.6.4.11 ISI Registers

Table 30 below details the ISI configuration registers. Note that some of these registers are also used by other blocks in the SCB.

Table 30. ISI configuration registers

Address Offset from SCB base	Register	#bits	Reset	Description
0x00	ISICntrl	5	0x2	ISI Control register
0x04	ISIId	4	0x1	ISIId for this SoPEC. A value of 0 indicates the device is an ISIMaster. Note that the SoPEC resets to being an ISISlave and that 0xF (the broadcast ISIId) is an illegal value and should not be written to this register.
0x08	NumRetries	4	0x02	Number of retransmissions to attempt in response to a NAK before aborting a long packet transmission
0x0C	ISIPingSchedule0	14	0x0000	Denotes which ISIIds will be receive ping packets. Note that bit0 refers to ISIId1, bit1 to ISIId2...bit13 to ISIId14.
0x10	ISIPingSchedule1	14	0x0000	As per *PingSchedule0*
0x14	ISIPingSchedule2	14	0x0000	As per *PingSchedule0*
0x18	ISITotalPeriod	4	0xF	Reload value of the ISITotalPeriod counter
0x1C	ISILocalPeriod	4	0xF	Reload value of the ISILocalPeriod counter
0x20	ISIStatus	6	0x00	ISI Status register. This register is **ReadOnly**.
0x24	ISIMask	6	0x00	ISI Interrupt Mask register
0x30 - 0x4C	CPUISITxBuff	32	n/a	32-byte CPUISI transmit buffer
0x50	CPUISITxBuffCntrl	13	0x0000	Control register for the CPUISI transmit buffer
0x60 - 0x7C	CPUISIRxBuff	32	n/a	32-byte ISI receive buffer. This is the half of the double buffer that contains the oldest data.
0x80	ISIRxBuffDest	1	0x0	Only one of the CPU and the DMA manager is allowed to empty the receive buffer at any time. 1 = CPU will empty the receive buffer 0 = DMA manager will empty the receive buffer

12.6.4.11.1 ISI control register

The ISICntrl register is described in Table 31 below. Note that the reset value of this register allows the SoPEC to automatically become an ISIMaster (*AutoMasterEnable* = 1) if any USB packets are received on

endpoints 2-4. On becoming an ISIMaster the *ISIId* register is set to 0, the *TxEnable* bit of the *ISICntrl* register is set and any USB or CPU packets destined for other ISI devices are transmitted. The CPU can override this capability at any time by clearing the *AutoMasterEnable* bit. Automatic ping operation can only be enabled by the CPU as the reset values of the *PingScheduleN* registers are all 0 and neither DMA channel is automatically configured.

Table 31. ISICntrl register

Field Name	bit(s)	Description
TxEnable	0	Enables ISI transmission of long or ping packets. This is cleared by transmit errors and so needs to be restarted by the CPU. Note that ACKs may still be transmitted when this bit is 0. 1 = Transmission enabled 0 = Transmission disabled
RxEnable	1	Enables ISI reception. This is cleared by receive errors and so needs to be restarted by the CPU. 1 = Reception enabled 0 = Reception disabled
AutoPingEnable	2	Enables auto-ping operation 1 = auto-ping enabled 0 = auto-ping disabled
AutoMasterEnable	3	Enables the device to automatically become the ISIMaster if activity is detected on USB endpoints2-4. 1 = auto-master operation enabled 0 = auto-master operation disabled

12.6.4.11.2 ISI status register

The *ISIStatus* register is read-only to the CPU. Status bits are set by the relevant condition occurring and are cleared by writing to either the *TxEnable* or *RxEnable* bits of the *ISICntrl* register or the *CPUISITx-Buff*.

Table 32. ISIStatus register

Field Name	bit(s)	Description
FrameError	0	Framing error detected in the received packet. This can be caused by an incorrect *Start* or *Stop* field or by bit stuffing errors
RxError	1	A CRC error or flow control condition was detected in *NumRetries*+1 successive packets (excluding ping packets)
RxBuffFull	2	There is no space remaining in the receive double buffer
RxBuffOverflow	3	An overflow has occurred in the ISI receive buffer and a packet had to be dropped.
CPUISITxBuffEmpty	4	The CPUISITxBuff is empty
TxError	5	Transmission error. Receiving ISI device would not accept the transmitted packet. Only set after *NumRetries* unsuccessful retransmissions (excluding ping packets).

12.6.4.11.3 ISI mask register

An interrupt will be generated in an edge sensitive manner i.e the ISI will generate an *isi_icu_irq* pulse each time a status bit goes high and the corresponding bit of the *ISIMask* register is enabled.

Table 33. ISIMask register

Field Name	bit(s)	Description
FrameErrorIntEn	0	Interrupt enable mask bit for the FrameError status bit
RxErrorIntEn	1	Interrupt enable mask bit for the RxError status bit
RxBuffFullIntEn	2	Interrupt enable mask bit for the RxBuffFull status bit
RxBuffOverflowIntEn	3	Interrupt enable mask bit for the RxBuffOverflow status bit
CPUISITxBuffEmpty-IntEn	4	Interrupt enable mask bit for the CPUISITxBuffEmpty status bit
TxErrorIntEn	5	Interrupt enable mask bit for the TxError status bit

12.6.4.11.4 CPUISITxBuffCntrl register

The *CPUISITxBuffCntrl* register contains the header field for the packet in the CPUISI transmit buffer. Writing to this buffer validates the contents of the CPUISI transmit buffer i.e. each time the CPU places a packet in the CPUISI transmit buffer it must write the packet header to this register to initiate its transfer in to the SCB transmit buffer (see section 12.7). Note that the CPU is responsible for toggling the sequence bit of any long packets it wishes to transmit. The *CPUISITxBuffEmpty* status bit will be set when *CPUTx-PktSize* bytes have been transferred to the SCB transmit buffer.

Table 34. CPUISITxBuffCntrl register

Field Name	bit(s)	Description
PktDesc	2:0	PktDesc field (as per Table 29) for the packet currently in the CPU-ISI transmit buffer.
DestISISubId	3	Indicates which DMAChannel of the target SoPEC the data in the CPUISI transmit buffer is destined for: 0 = DMAChannel0 1 = DMAChannel1
DestISIId	7:4	Denotes the ISIId of the target SoPEC as per Table 35

12.7 SCB MAPPING

In order to support maximum flexibility when moving data through a multi-SoPEC system it is possible to map any USB endpoint onto either DMAChannel within any SoPEC in the system. A logical view of the

SCB is shown in Figure 37. This view differs from the likely implementation but it allows for a clearer depiction of data movement within the SCB.

SCB
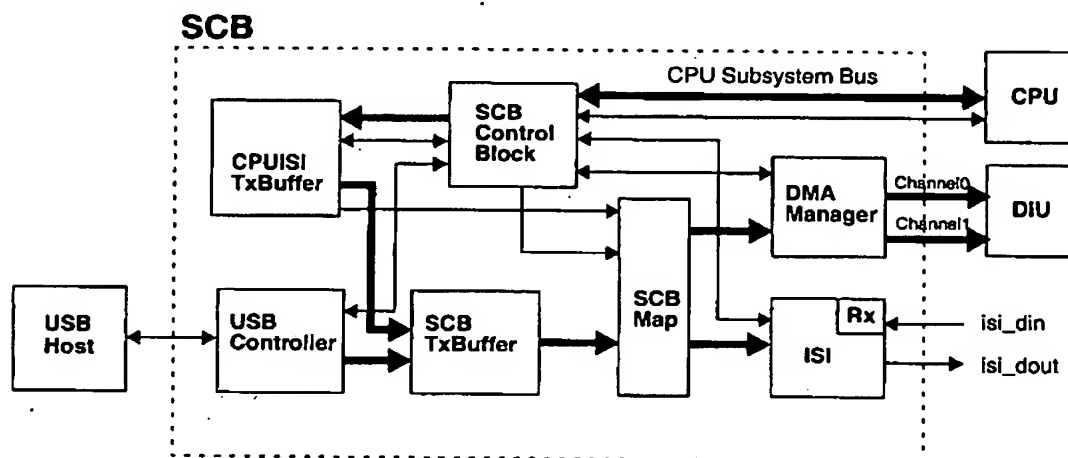
Figure 37. SCB logical view

The SCB map, and indeed the SCB itself is based around the concept of an ISIId and an ISISubId. Each SoPEC in the system has a unique ISIId and two ISISubIds, namely ISISubId0 and ISISubId1. We use the convention that ISISubId0 corresponds to DMAChannel0 in each SoPEC and ISISubId1 corresponds to DMAChannel1. The naming convention for the ISIId is shown in Table 35 below and this would correspond to a multi-SoPEC system such as that shown in Figure 27. We use the term ISIId instead of SoPECId to avoid confusion with the unique ChipID used to create the SoPEC_id and SoPEC_id_key (see chapter 17 and [9] for more details).

Table 35. ISIId naming convention

ISIId	SoPEC to which it refers
0	ISIMaster (typically a SoPEC connected to the host via USB1.1)
1 - 14	ISISlave1-14
15	Broadcast ISIId

The combined ISIId and ISISubId therefore allow us to address any DMAChannel in the system. The ISI, DMA manager and SCB map hardware use the ISIId and ISISubId to handle the different data streams that are active in a multi-SoPEC system as does the software running on the CPU of each SoPEC. In this document we will identify DMAChannels as *ISIx.y* where *x* is the ISIId and *y* is the ISISubId. Thus ISI2.1 refers to DMAChannel1 of ISISlave2. Any data sent to a broadcast channel, i.e. ISI15.0 or ISI15.1, are received by every ISI device in the system including the ISIMaster (which may be an ISI-Bridge).

The USB controller and software stacks however have no understanding of the ISIId and ISISubId but the Silverbrook printer driver software running on the host PC does make use of the ISIId and ISISubId. USB is simply used as a data transport - the mapping of USB endpoints onto ISIId and SubId is communicated from the host PC Silverbrook code to the SoPEC Silverbrook code through USB control (or possibly bulk data) messages i.e. the mapping information is simply data payload as far as USB is concerned. The code running on SoPEC is responsible for parsing these messages and configuring the SCB accordingly.

The use of just two DMAChannels places some limitations on what can be achieved without software intervention. For every SoPEC in the system there are more potential sources of data than there are sinks. For example an ISISlave could receive both control and data messages from the ISIMaster SoPEC in addition to control and data from the host, either specifically addressed to that particular ISISlave or over the broadcast ISI channel. However all ISISlaves only have two possible data sinks, i.e. the two DMAChannels. Another example is the ISIMaster in a multi-SoPEC system which may receive control messages from each SoPEC in addition to control and data information from the host (e.g. over USB). In this case all of the control messages are in contention for access to DMAChannel0. We resolve these potential conflicts by adopting the following conventions:

1) **Control messages may be interleaved in a memory buffer:** The memory buffer that the DMAChannel0 points to should be regarded as a central pool of control messages. Every control message must contain fields that identify the size of the message, the source and the destination of the control message. Control messages may therefore be multiplexed over a DMAChannel which allows several control message sources to address the same DMAChannel. Furthermore, if SoPEC-type control messages contain source and destination fields it is possible for the host to send control messages to individual SoPECs over the ISI15.0 broadcast channel.

2) **Data messages should not be interleaved in a memory buffer:** As data messages are typically part of a much larger block of data that is being transferred it is not possible to control their contents in the same manner as is possible with the control messages. Furthermore we do not want the CPU to have to perform reassembly of data blocks. Data messages from different sources cannot be interleaved over the same DMAChannel - the SCB map must be reconfigured each time a different data source is given access to the DMAChannel.

3) **Every reconfiguration of the SCB map requires the exchange of control messages:** The only active SCB map in a multi-SoPEC system is the SCB map in the ISIMaster as all ISISlaves automatically send data addressed to themselves to either DMAChannel0 or 1 i.e. the ISI is the only source of incoming data in an ISISlave. The ISIMaster's SCB map reset state is shown in Figure 39 and any subsequent modifications require the exchange of control messages between the ISIMaster and the host. As the host is expected to control the movement of data in any SoPEC system it is anticipated that all changes to the SCB map will be performed in response to a request from the host. While the ISIMaster could autonomously reconfigure the SCB map (this is entirely up to the software running on the ISIMaster) it should not do so without informing the host in order to avoid data being misrouted.

An example of the above conventions in operation is worked through in section 12.7.2.

12.7.1   Host PC to ISIMaster SoPEC communication

When considering SCB map configurations we always assume that the ISIMaster is a SoPEC device, in particular the SoPEC connected to the USB bus (and receiving data on USB endpoint 2, 3 or 4), rather than an ISI-Bridge chip. ISI-Bridge chips are likely to have something similar to an SCB map and the following information should broadly apply to an ISI-Bridge but we focus here on an ISIMaster SoPEC for clarity.

As the ISIMaster SoPEC represents the printer on the USB bus it is required by the USB specification to have a dedicated control endpoint, EP0. At boot time the ISIMaster SoPEC will also require a bulk data endpoint to facilitate the transfer of program code from the host PC. The simplest SCB map configuration, i.e. for a single stand-alone SoPEC, is sufficient for host to ISIMaster SoPEC communication and is shown in Figure 38. In this configuration all USB control information exchanged between the host and SoPEC over EP0 (which is the only bidirectional USB endpoint). SoPEC specific control information (printer status, DNC info etc.) is also exchanged over EP0.

All packets sent to the host from SoPEC over EP0 must be written into the EP0 FIFO by the CPU. All packets sent from the host to SoPEC can be placed in DRAM by the DMA Manager (as is usually the case) or read directly by the CPU. This asymmetry is because in a multi-SoPEC environment the CPU will

need to examine all incoming control messages (i.e. messages that have arrived over DMAChannel0) to ascertain their source and destination (i.e. they could be from an ISISlave and destined for the host) and so the additional overhead in having the CPU move the short control messages to the EP0 FIFO is relatively small. Furthermore we wish to avoid making the SCB more complicated than necessary, particularly when there is no significant performance gain to be had as the control traffic will be relatively low bandwidth.

The above mechanisms are appropriate for the types of communication outlined in sections 12.4.1.1 through 12.4.1.4
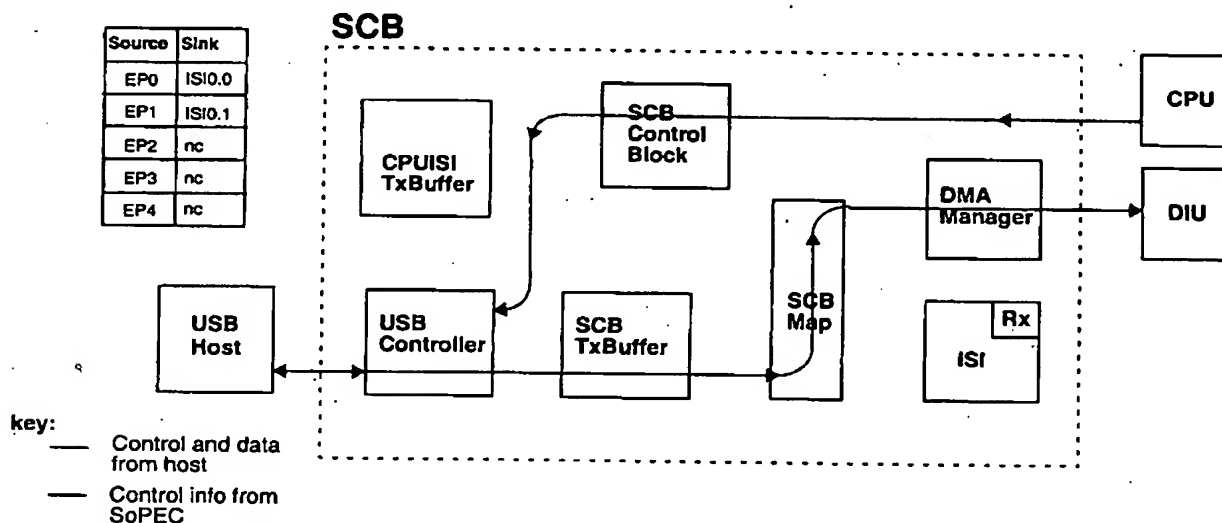
Figure 38. Single SoPEC SCB map configuration and dataflow

12.7.2 Broadcast communication

An SCB configuration for broadcast communication is shown in Figure 39. This particular configuration is also the default, post power-on reset, configuration for the ISIMaster SoPEC. USB endpoints EP2 and EP3 are mapped onto ISISubID0 and ISISubId1 of ISIId15 (the broadcast ISIId channel). EP0 is used for control messages as before and EP1 is a bulk data endpoint for the ISIMaster SoPEC. Depending on what is convenient for the boot loader software, EP1 may or may not be used during the initial program download, but EP1 is highly likely to be used for compressed page or other program downloads later. For this reason it is part of the default configuration. In this setup the USB device configuration will take place, as it always must, by exchanging messages over the control channel (EP0).

One possible boot mechanism is where the host PC sends the bootloader1 program code to all SoPECs by broadcasting it over EP3. Each SoPEC in the system then authenticates and executes the bootloader1 program. The ISIMaster SoPEC then polls each ISISlave (over the ISIx.0 channel). Each ISISlave ascertains its ISIId by sampling the particular GPIO pins required by the bootloader1 and reporting its presence and status back to the ISIMaster. The ISIMaster then passes this information back to the host over EP0. Thus both the host and the ISIMaster have knowledge of the number of SoPECs, and their ISIIds, in the system. The host may then reconfigure the SCB map to better optimise the SCB resources for the particular multi-SoPEC system. This could involve simplifying the default configuration to a single SoPEC system (Figure 38) or remapping the broadcast channels onto DMAChannels in individual ISISlaves.

Source	Sink
EP0	ISI0.0
EP1	ISI0.1
EP2	ISI15.0
EP3	ISI15.1
EP4	ISI1.1

key:
⎯ Control and data from host
⎯ Control info from SoPEC to host
⎯ Control Info from SoPEC to ISISlaves
⎯ Control info from ISISlaves to SoPEC
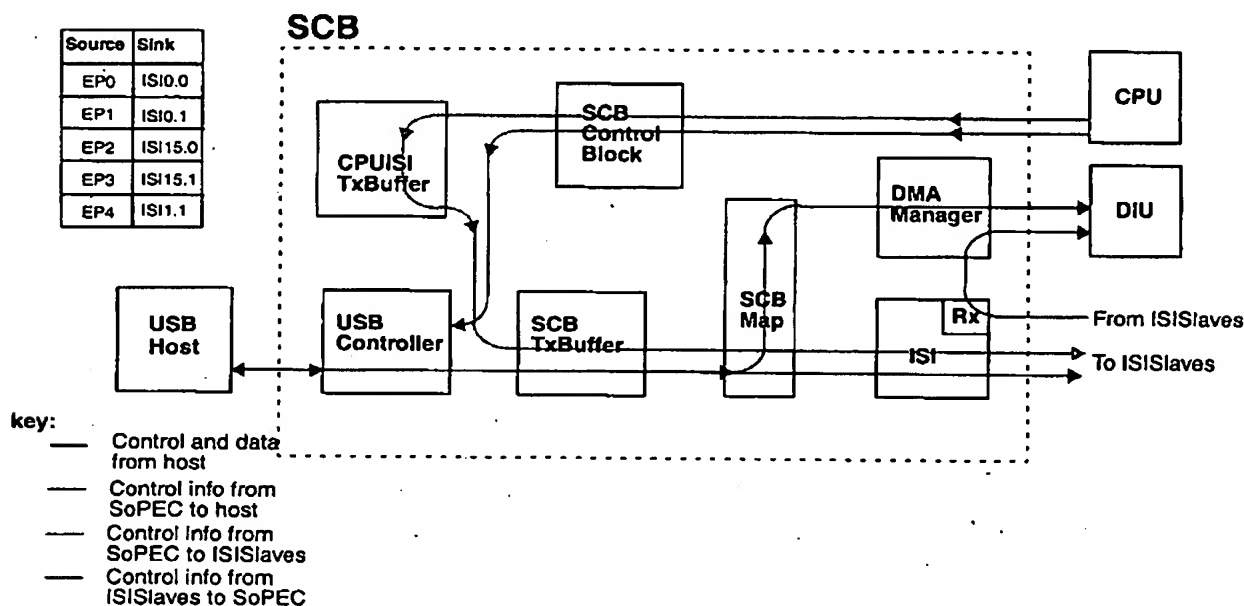
Figure 39. Default SoPEC SCB map configuration and dataflow

The following steps are required to reconfigure the SCB map from the system depicted in Figure 39 to one where EP3 is mapped onto ISI1.0:

1.) The host PC sends a control message(s) to the ISIMaster SoPEC requesting that USB EP3 be remapped to ISI1.0

2) The ISIMaster SoPEC sends a control message to the host PC informing it that EP3 has now been mapped to ISI1.0 (and therefore the host knows that the previous mapping of ISI15.1 is no longer available through EP3).

3) The host may now send control messages directly to ISISlave1 without requiring any CPU intervention on the ISIMaster SoPEC

12.7.3   Host PC - ISISlave SoPEC communication

The default post-boot (as opposed to post-reset) SCB map configuration for an ISISlave SoPEC is to have all USB endpoints unconnected. The ISI automatically forwards any data addressed to it (including broadcast data) to the DMA with the appropriate ISISubId. If the ISIMaster is configured correctly (e.g. when the ISIMaster is a SoPEC, and that SoPEC's SCB map is configured correctly) then data sent from the host destined for an ISISlave will be transmitted on the ISI with the correct address. If the ISISlave has data to send to the host it must do so by sending a control message to the ISIMaster identifying the host as the intended recipient. It is then the ISIMaster's responsibility to forward this message to the host.

With this configuration the host can communicate with the ISISlave via broadcast messages only and this is the mechanism by which the bootloader1 program is downloaded. The ISISlave is unable to communicate with the host (or the ISIMaster) until the bootlloader1 program has successfully executed and the ISISlave has determined what its ISIId is. After the bootloader1 program (and possibly other programs)

has executed the SCB map of the ISIMaster may be reconfigured to reflect the most appropriate topology for the particular multi-SoPEC system it is part of.

All communication from an ISISlave to host is achieved by sending messages via the ISIMaster. The ISISlave can never initiate communication to the host. If an ISISlave wishes to send a message to the host it may do one of two things: (a) wait until it is polled by the ISIMaster or (b) indicate in its ISI acknowledgement packet (sent in response to the reception of an ISI packet specifically addressed to that ISISlave) that it has a message to send. When the ISIMaster receives the message from the ISISlave it first examines it to determine the intended destination and will then copy it into the EP0 FIFO for transmission to the host. The software running on the ISIMaster is responsible for any arbitration between messages from different sources (including itself) that are all destined for the host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.4.1.5 and 12.4.1.6.

12.7.4   ISIMaster - ISISlave communication

All ISIMaster - ISISlave communication takes place over the ISI. Immediately after reset this can only be by means of broadcast messages. Once the bootloader1 program has successfully executed on all SoPECs in a multi-SoPEC system the ISIMaster can communicate with each SoPEC on an individual basis.

If an ISISlave wishes to send a message to the ISIMaster it may do so in response to a ping packet from the ISIMaster. When the ISIMaster receives the message from the ISISlave it must interpret the message to determine if the message contains information required to be sent to the host. In the case of the ISIMaster being a SoPEC, software will transfer the appropriate information into the EP0 FIFO for transmission to the host.

The above mechanisms are appropriate for the types of communication outlined in sections 12.4.2.3 and 12.4.2.4.

12.7.5   ISISlave - ISISlave communication

ISISlave to ISISlave communication is expected to be limited to two special cases: (a) when the PrintMaster is not the ISIMaster and (b) when a storage SoPEC is used. When the PrintMaster is not the ISIMaster then it will need to send control messages (and receive responses to these messages) to other ISISlaves. When a storage SoPEC is present it may need to send data to each SoPEC in the system. All ISISlave to ISISlave communication will take place in response to ping messages from the ISIMaster.

12.7.6   SCB Map configuration registers

The SCB map is configured by mapping a USB endpoint on to a data sink. This is performed on a endpoint basis i.e. each endpoint has a configuration register to allow its data sink be selected. Mapping an endpoint on to a data sink does not initiate any data flow - each endpoint/data sink needs to be enabled by writing to the appropriate configuration registers in the USB controller/ ISI / DMA manager.

Table 36. SCB Map configuration registers

Address/ Offset from SCB base	Register	#bits	Reset	Description
0x100	USBEP0Dest	7	0x20	This register determines which of the data sinks the data arriving in EP0 should be routed to.
0x104	USBEP1Dest	7	0x21	Data sink for USB EP1
0x108	USBEP2Dest	7	0x3E	Data sink for USB EP2

Table 36. SCB Map configuration registers

Address Offset from SCB base	Register	#bits	Reset	Description
0x10C	USBEP3Dest	7	0x3F	Data sink for USB EP3
0x110	USBEP4Dest	7	0x23	Data sink for USB EP4

The same encoding is used for each of the *USBEPnDest* configuration registers and is described in Table 37. The *ISIId* register (see Table 30) allows the SCB map to identify data that should be routed to the DMA Manager as well as, or instead of, to the ISI. The SCB map therefore does not need special fields to identify the DMAChannels on the ISIMaster SoPEC as this is taken care of by the SCB hardware. Thus the *USBEP0Dest* and *USBEP1Dest* registers should be programmed with 0x20 and 0x21 (for ISI0.0 and ISI0.1) respectively to ensure data arriving on these endpoints is moved directly to DRAM.

Table 37. USBEPnDest register

Field Name	#bits	Description
DestISISubId	0	Indicates which DMAChannel of the target SoPEC the endpoint is mapped onto: 0 = DMAChannel0 1 = DMAChannel1
DestISIId	4:1	Denotes the ISIId of the target SoPEC as per Table 35
ChannelEn	5	Enable bit for the DMAChannel: 0 = Channel disabled 1 = Channel enabled
SequenceBit	6	Sequence bit for packets going from USBEPn to DestISIId.Dest-ISISubId. Every CPU write to this register initialises the value of the sequence bit and this is subsequently updated by the ISI after every successful long packet transmission.

A SoPEC ISIMaster should map as many USB endpoints, under the control of the host, as are required for the multi-SoPEC system it is part of. As already mentioned this mapping may be dynamically reconfigured.

12.7.7   SCB transmit buffer arbitration

When the SCB transmit buffer has been emptied the SCB control logic will immediately seek to refill it. As there may be data waiting in a USB endpoint FIFOs and in the CPUISI transmit buffer it may be necessary to arbitrate between these data sources. This arbitration is controlled by the *SCBTxBuffArb* register which contains a high priority bit for both the CPU and the USB. If only one of these bits is set then the corresponding source always has priority. Note that if the CPU is given absolute priority over the USB the software filling the CPUISI transmit buffer needs to ensure that sufficient USB traffic is allowed through. If both bits of the *SCBTxBuffArb* have the same value then arbitration will take place on a round robin basis.

As the speed at which the SCB transmit buffer can be emptied is at least 5 times greater than it can be filled by USB traffic the double buffers used for each USB endpoint will not overflow using the above scheme in normal operation. There are a number of scenarios which could lead to the USB endpoints being temporarily blocked such as the CPU having priority, retransmissions on the ISI bus, channels being enabled (cf. the *ChannelEn* bit of the *USBEPnDest* register) with data already in their associated endpoint FIFOs or

short packets being sent on the USB. Care should be taken to ensure that the USB bandwidth is efficiently utilised at all times.

12.7.8 SCB Control Block

The SCB control block is responsible for coordinating access to and between the various sub-blocks in the SCB. This includes translating between the CPU subsystem bus and the USB native bus protocol, moving data from the USB endpoint FIFOs into the SCB transmit buffer, moving data from the CPUISI transmit buffer into the SCB transmit buffer and arbitrating between the CPU and itself for access to the SCB sub-blocks.

Table 38. SCB control block configuration registers

Address Offset from SCB base	Register	#bits	Reset	Description
0x120	WakeupEnable	2	0x0	This register is used to gate the propagation of the USB and ISI reset signals to the CPR block. Active high. WakeUpEnable[0]: *usb_cpr_reset_n* control WakeUpEnable[1]: *isi_cpr_reset_n* control
0x124	SCBTxBuffArb	2	0x0	Determines which source has priority when contention arises in filling the SCBTxBuffer. When a bit is set priority is given to the relevant source. SCBTxBuffArb[0]: CPU priority SCBTxBuffArb[1]: USB priority
0x128	SCBDebugSel	10	0x000	Contains address of the register selected for debug observation as it would appear on cpu_adr[11:2] The contents of the selected register are output in the *scb_cpu_data* bus while *cpu_scb_sel* is low and *scb_cpu_debug_valid* is asserted to indicate the debug data is valid. It is expected that a number of pseudo-registers will be made available for debug observation and these will be outlined with the implementation details.

12.8 DMA MANAGER

The DMA manager manages the flow of data between the SCB and the embedded DRAM. Whilst the CPU could be used for the movement of data in a USB1.1 enabled SoPEC a DMA manager is a more efficient solution as it will handle data in a more predictable fashion with less latency and requiring less buffering. Furthermore a DMA manager is required to support the ISI transfer speed and to ensure that the SoPEC could be used with a high speed ISI-Bridge chip in the future.

The DMA manager uses two independent channels, one for each ISISubId, to control the movement of data. Both DMAChannels only support write operation and can transfer data from any USB endpoint and from the ISI receive buffer. Data is moved at the soonest opportunity to do so and is always moved in 256-bit slices as required by the DIU. When it is not possible to use a 256-bit slice of data (e.g. at the end of a packet or for a short packet) the DMA manager will still use 256-bit access to the DIU. This means that for a DIU write (data incoming to the SoPEC) the DMA manager will pad the valid data with zeroes until a 256-bit slice has been filled.

The DMA manager handles all issues relating to byte/word/longword address alignment, data endianness and transaction scheduling. It arbitrates between data arriving from the ISI and data arriving from a USB

endpoint on a round robin basis. The greater guaranteed bandwidth available to the DMA manager (50 Mbit/s at the time of writing but this may need to be increased especially if a 4-wire ISI bus is used. See section 20.6 for more details) ensures that the DMA manager is non-blocking.

While the DMA manager performs the work of moving data the CPU controls the destination and relative timing of dataflows to and from the DRAM. The management of the DRAM data buffers requires the CPU to have accurate and timely visibility of both the DMA and PEP memory usage. In other words when the PEP has completed processing of a page band the CPU needs to be aware of the fact that an area of memory has been freed up to receive incoming data. The management of these buffers may also be performed by the host.

12.8.1   Circular buffer operation

The DMA manager supports the use of circular buffers for both DMAChannels. Each circular buffer is controlled by 5 registers: *DMAnBottomAdr, DMAnTopAdr, DMAnMaxAdr, DMAnCurrWPtr* and *DMAnIntAdr*. The operation of the circular buffers is shown in Figure 40 below.

Key:

☐ Free buffer space

▓ Filled buffer space (unprocessed data)

▨ Buffer space filled since last write to the DMAnIntAdr/DMAnMaxAdr registers
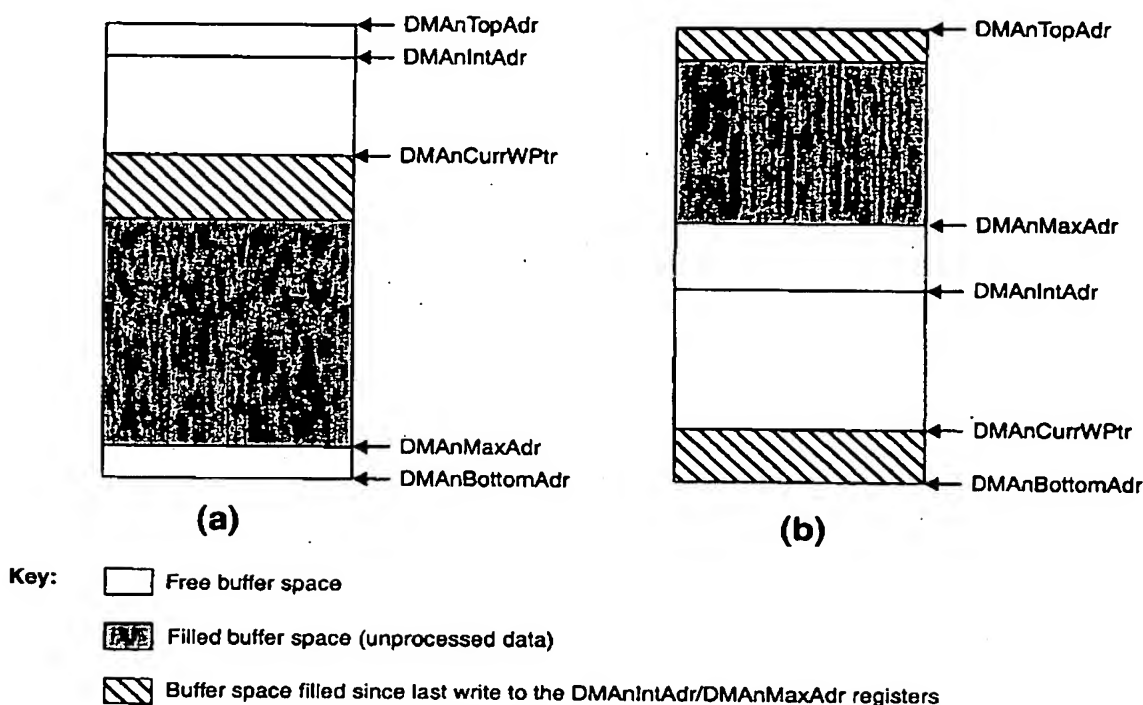
Figure 40. Circular buffer operation

Here we see two snapshots of the status of a circular buffer with (b) occurring sometime after (a) and some CPU writes occurring in between (a) and (b). These CPU writes are most likely to be as a result of a finished band interrupt (which frees up buffer space) but could also have occurred in a DMA interrupt service routine resulting from *DMAnIntAdr* being hit. The DMA manager will continue filling the free buffer space depicted in (a), advancing the *DMAnCurrWPtr* after each write to the DIU. Note that the *DMACurrWPtr* register always points to the next address the DMA manager will write to. When the DMA manager

reaches the address in *DMAnIntAdr* (i.e. *DMACurrWPtr* = *DMAnIntAdr*) it will generate an interrupt if the *DMAnIntAdrMask* bit in the *DMAMask* register is set. The purpose of the *DMAnIntAdr* register is to alert the CPU that data (such as a control message or a page or band header) has arrived that it needs to process. The interrupt routine servicing the DMA interrupt will change the *DMAnIntAdr* value to the next location that data of interest to the CPU will have arrived by.

In the scenario shown in Figure 40 the CPU has determined (most likely as a result of a finished band interrupt) that the filled buffer space in (a) has been freed up and is therefore available to receive more data. The CPU therefore moves the *DMAnMaxAdr* to the end of the section that has been freed up and moves the *DMAnIntAdr* address to an appropriate offset from the *DMAnMaxAdr* address. The DMA manager continues to fill the free buffer space and when it reaches the address in *DMAnTopAdr* it wraps around to the address in *DMAnBottomAdr* and continues from there. DMA transfers will continue indefinitely in this fashion until the DMA manager reaches the address in the *DMAnMaxAdr* register.

The circular buffer is initialised by writing the top and bottom addresses to the *DMAnTopAdr* and *DMAn-BottomAdr* registers, writing the start address (which does not have to be the same as the *DMAnBottomAdr* even though it usually will be) to the *DMAnCurrWPtr* register and appropriate addresses to the *DMAnIntAdr* and *DMAnMaxAdr* registers. The DMA operation will not commence until a 1 has been written to the relevant bit of the *DMAChanEn* register.

While it is possible to modify the *DMAnTopAdr* and *DMAnBottomAdr* registers after the DMA has started it should be done with caution. The *DMAnCurrWPtr* register should not be written to while the DMAChannel is in operation. DMA operation may be stalled at any time by clearing the appropriate bit of the *DMAChanEn* register or by disabling an SCB mapping or ISI receive operation.

12.8.2   DMA manager DRAM bandwidth requirements

The DIU must guarantee the SCB enough bandwidth to ensure that neither a USB endpoint FIFO nor the ISI receive buffer can overrun. For example, to facilitate bursty 32 Mbit/s transfers a SoPEC with a 64-byte ISI receive buffer would need to be able to transfer 256 bits every 1280 cycles (@160 MHz). This is in addition to the USB transactions targeted at the ISIMaster SoPEC which may be in the region of 8-9 Mbit/s. While USB has a backpressure mechanism SoPEC should strive to obtain optimum USB bandwidth utilization and so USB backpressuring should only be used as a last resort. The DIU currently guarantees 50 Mbit/s to the SCB and more bandwidth will be available when other DIU requestors do not take their slots. This is sufficient for the SCB's requirements.

12.8.3   DMA manager configuration registers

All of the circular buffer registers are 256-bit word aligned as required by the DIU. The *DMAnBottomAdr* and *DMAnTopAdr* registers are inclusive i.e. the addresses contained in those registers form part of the circular buffer. The *DMAnCurrWPtr* always points to the next location the DMA manager will write to so interrupts are generated whenever the DMA manager reaches the address in either the *DMAnIntAdr* or

DMAnMaxAdr registers rather than when it actually writes to these locations. It therefore cannot write to the location in the *DMAnMaxAdr* register.

Table 39. DMA Manager Configuration Registers

Address Offset from SCB base	Register	#bits	Reset	Description
0x200	DMA0BottomAdr	17	0x0_0000	The 256-bit aligned DRAM address of the bottom of the circular buffer serviced by DMAChannel0
0x204	DMA0TopAdr	17	0x0_0000	The 256-bit aligned DRAM address of the top of the circular buffer serviced by DMAChannel0
0x208	DMA0CurrWPtr	17	0x0_0000	The 256-bit aligned DRAM address of the next location DMAChannel0 will write to. This register is set by the CPU at the start of a DMA operation and dynamically updated by the DMA manager during the operation.
0x20C	DMA0IntAdr	17	0x0_0000	The 256-bit aligned DRAM address of the location that will trigger an interrupt when reached by DMAChannel0 buffer.
0x210	DMA0MaxAdr	17	0x0_0000	The 256-bit aligned DRAM address of the last free location in the DMAChannel0 circular buffer. The DMAChannel0 transfers will stop when it reaches this address.
0x214	DMA0SeqBit	1	0x0	Sequence bit for DMAChannel0. This bit may be initialised by the CPU but is updated by the ISI each time an error-free long packet is received.
0x218	DMA1BottomAdr	17	0x0_0000	The 256-bit aligned DRAM address of the bottom of the circular buffer serviced by DMAChannel1
0x21C	DMA1TopAdr	17	0x0_0000	The 256-bit aligned DRAM address of the top of the circular buffer serviced by DMAChannel1
0x220	DMA1CurrWPtr	17	0x0_0000	The 256-bit aligned DRAM address of the next location DMAChannel1 will write to. This register is set by the CPU at the start of a DMA operation and dynamically updated by the DMA manager during the operation.
0x224	DMA1IntAdr	17	0x0_0000	The 256-bit aligned DRAM address of the location that will trigger an interrupt when reached by DMAChannel1 buffer.
0x228	DMA1MaxAdr	17	0x0_0000	The 256-bit aligned DRAM address of the last free location in the DMAChannel1 circular buffer. The DMAChannel1 transfers will stop when it reaches this address.
0x22C	DMA1SeqBit	1	0x0	Sequence bit for DMAChannel1. This bit may be initialised by the CPU but is updated by the ISI each time an error-free long packet is received.
0x230	DMAChanEn	2	0x0	Enable DMA operation on a per channel basis. Active high. DMAChanEn[0]: Enable DMAChannel0 DMAChanEn[1]: Enable DMAChannel1

Table 39. DMA Manager Configuration Registers

Address Offset from SCB base	Register	#bits	Reset	Description
0x234	DMAStatus	4	0x0	DMA status register. See section 12.8.3.1. This register is **ReadOnly**.
0x238	DMAMask	4	0x0	DMA mask register. See section 12.8.3.2

12.8.3.1 DMAStatus register

The contents of the DMAStatus register are read-only to the CPU. The status bits are not sticky bits i.e. they reflect the 'live' status of the channel. Status bits may only be cleared by writing to the relevant *DMAnIntAdr* or *DMAnMaxAdr* register.

Table 40. DMA Status Register

Field Name	bit(s)	Description
DMAChannel0IntAdrHit	0	DMAChannel0 has reached the address contained in the *DMA0IntAdr* register
DMAChannel0MaxAdrHit	1	DMAChannel0 has reached the address contained in the *DMA0MaxAdr* register
DMAChannel1IntAdrHit	2	DMAChannel1 has reached the address contained in the *DMA1IntAdr* register
DMAChannel1MaxAdrHit	3	DMAChannel1 has reached the address contained in the *DMA1MaxAdr* register

12.8.3.2 DMAMask register

All bits of the DMAMask are both readable and writable by the CPU. The DMA manager cannot alter the value of this register. All interrupts are edge sensitive i.e the DMA manager will generate a *dma_icu_irq* pulse each time a status bit goes high and the corresponding mask bit is enabled.

Table 41. DMA Manager Mask Register

Field Name	bit(s)	Description
DMAChannel0IntAdrHitMask	0	1 = Generate an interrupt when the DMAChannel0IntAdrHit status bit goes high 0 = Do not generate an interrupt when the DMAChannel0IntAdrHit status bit goes high
DMAChannel0MaxAdrHitMask	1	1 = Generate an interrupt when the DMAChannel0MaxAdrHit status bit goes high 0 = Do not generate an interrupt when the DMAChannel0MaxAdrHit status bit goes high
DMAChannel1IntAdrHitMask	2	As per DMAChannel0IntAdrHitMask
DMAChannel1MaxAdrHitMask	3	As per DMAChannel0MaxAdrHitMask

12.9 SCB IMPLEMENTATION

This section is still a work in progress - the information here should be ignored as it refers to an earlier version of the SCB
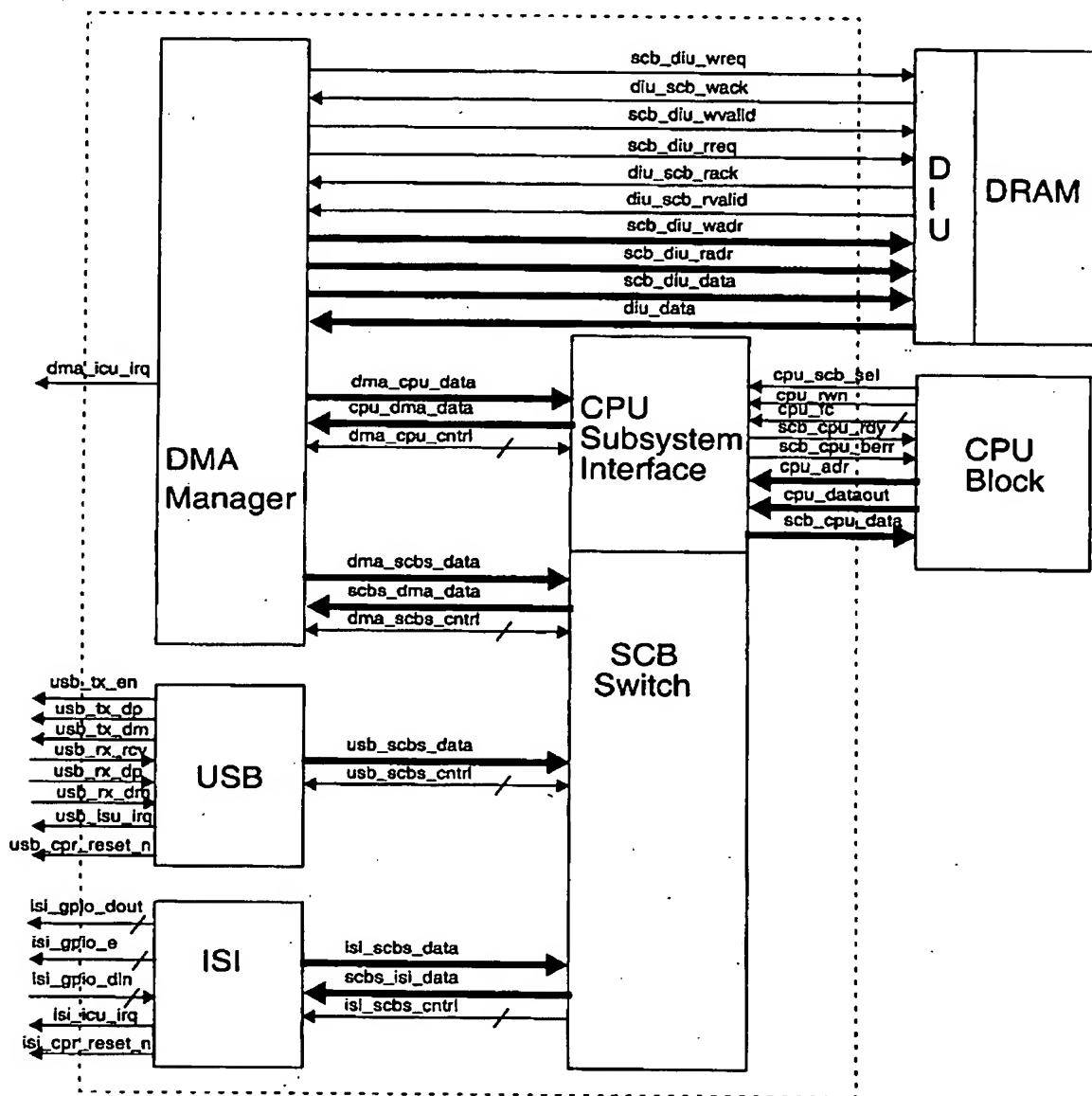
Characteristics of the data channels:

USB: Packets should be moved sequentially out of the endpoint FIFOs. The USB is the slowest component in the SCB but its bandwidth is most precious. However both the DMA and ISI can transfer data (50 and 40 Mbps respectively) much faster than the USB can receive data (12 Mbps peak rate) so no flow control problems will occur due to a speed mismatch. If one of the DMA or ISI data sinks becomes blocked or inactive then the USB controller will assert backpressure (by NAKing packets) when the double buffer for the associated endpoint is filled. Other endpoints will remain active in this scenario and the DMA and ISI will still be able to transfer data at their peak rates. The worst case scenario is when all endpoints have their double buffers filled (because all the data sinks had been blocked/disabled) and then all data sinks become available again. In this case the backlog will be fully cleared in 3 USB 64-byte packet times.

ISI: The ISI can support simultaneous reception and transmission of packets. ISI packets should be transferred sequentially in either direction. The ISI is expected to handle the packet header and trailer, if any is used for error detection, in both directions i.e. only raw payload data is routed through the SCB map.

DMA: The DMA channels are unidirectional but their direction, namely whether they are transferring data to or from DRAM, is programmable. Each DMA transaction to DRAM will be 256 bits wide but all 256 bits are not always valid. When a transfer of less than 256 bits is required the DMA manager pads the remaining bits in the 256-bit word with zeroes, in the case of a write to DRAM, or discards the unneccessary bits in the case of a DRAM read. Can we get by with single (256 bits each way or maybe even 256 bits in all ?) buffering for the DRAM manager ?
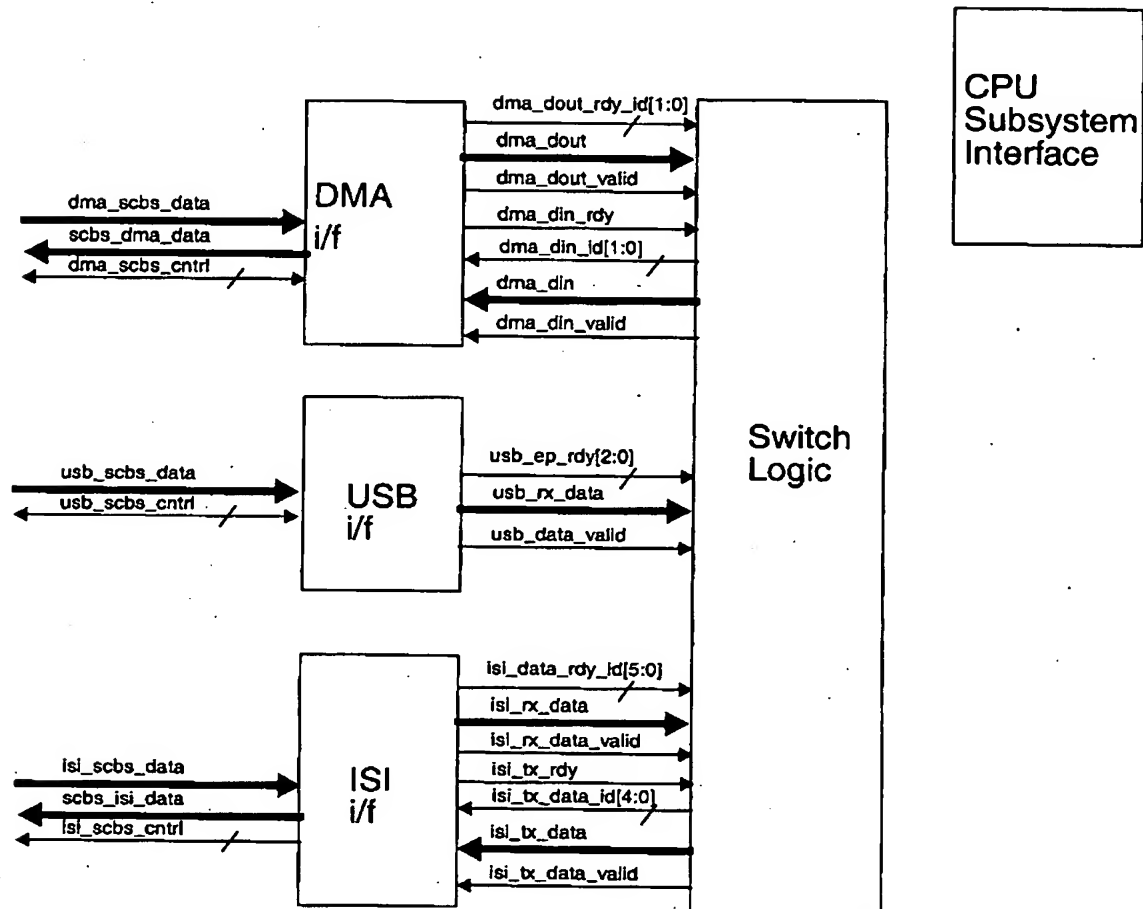
Figure 41. SCB Switch block diagram

SCB Switch pseudocode:

```
const no_data_sinks = 12

for i = 1 to no_data_sinks
    if (i <= 2) then
        sink_data is dma_din
        sink_rdy is dma_din_rdy
        sink_data_valid is dma_din_valid
        sink_id is dma_din_id
```

```
            else
                sink_data is isi_tx_data
                sink_rdy is isi_tx_rdy
                sink_data_valid is isi_tx_data_valid
                sink_id is isi_tx_data_id

        if (data_src_reg[i] != 0) then   // Each data sink has an associated data source
                                         // register. A non-zero value means the sink is enabled
            if ((data_src_reg[i] & 0xF0) == 0x10) then // A USB endpoint is the data source
                if ((usb_ep_rdy[4] == 1) AND (usb_ep_rdy[3:0] == data_src_reg[i][3:0])) then
                                            // there is data waiting in the EP FIFO
                    while ((usb_data_valid == 1) AND (sink_rdy == 1) AND clocktick)
                        sink_data = usb_rx_data
                        sink_data_valid = 1
                        if (i <= 2) then  // The sink is a DMAChannel
                            sink_id[1] = 1
                            sink_id[0] = i -1
                        else    // The sink is an ISI channel
                            sink_id[5] = 1
                            sink_id[4:0] = i -1
                else    // There is no data ready to go
                    sink_data_valid = 0

            elsif (data_src_reg & 0xF0) == 0x20) then   // The ISI is the data source
                if (isi_data_rdy_id[3:0] == data_src_reg[i][3:0]) then // there is data waiting
                                            // in the ISI receive FIFO for this ISISubId
                    while ((isi_rx_data_valid == 1) AND (sink_rdy == 1) AND clocktick)
                        sink_data = isi_rx_data
                        sink_data_valid = 1
                        if (i <= 2) then  // The sink is a DMAChannel
                            sink_id[1] = 1
                            sink_id[0] = i -1
                        else    // The sink is an ISI channel
                            sink_id[5] = 1
                            sink_id[4:0] = i -3
                else    // There is no data ready to go
                    sink_data_valid = 0

            elsif (data_src_reg & 0xF0) == 0x30) then   // The DMA is the data source
                if (dma_dout_rdy_id[0] == data_src_reg[i][0]) then // there is data waiting
                                            // in the relevant DMA buffer for this sink
                    while ((dma_dout_valid == 1) AND (sink_rdy == 1) AND clocktick)
                        sink_data = dma_dout
                        sink_data_valid = 1
                        if (i <= 2) then. // The sink is a DMA channel
                            sink_id[1] = 1
                            sink_id[0] = i -1
                        else    // The sink is an ISI channel
                            sink_id[5] = 1
                            sink_id[4:0] = i -3
                else    // There is no data ready to go
                    sink_data_valid = 0
```

The above pseudocode has a few shortcomings, particularly if all our data buses are not the same size, but it shows the basic functionality the switch is supposed to offer. The main loop of the pseudocode (for i = 1 to no_data_sinks) dictates what happens within one timeslot. The timeslots take as long as required to complete and loop around endlessly. The msb of the *usb_ep_rdy[4:0]*, *isi_data_rdy_id[5:0]* and *dma_dout_rdy_id[1:0]* signals is used to indicate that data is available in the relevant block.

13 General Purpose IO (GPIO)

13.1 OVERVIEW

The General Purpose IO block (GPIO) is responsible for control and interfacing of GPIO pins to the rest of the SoPEC system. It provides easily programmable control logic to simplify control of GPIO functions. In all there are 14 GPIO pins of which certain pins have special functions, their functions are detailed as:

- 4 Motor control IOs internally pulled down
- 4 General purpose high drive pulsed IOs capable of driving LEDs.
- 4 Open drain IOs used for LSS interfaces
- 2 Normal drive IOs used for the ISI interface in Multi-SoPEC mode

Each of the pins can be configured in either input or output mode, each pin is independently controlled. A programmable de-glitching circuit exists for all input pins. Each input is a schmidt trigger to increase noise immunity should the input be used without the de-glitch circuit. The mapping of the above functions and their alternate use in a slave SoPEC to GPIO pins is shown in Table 42 below.

Table 42. GPIO pin functionality

GPIO pins	Function
gpio[3:0]	Motor control pins / general purpose IO
gpio[7:4]	LED driver pins / general purpose IO
gpio[11:8]	LSS interface pins / general purpose IO
gpio[13:12]	ISI interface pins / general purpose IO

13.2 MOTOR CONTROL

The motor control pins can be directly controlled by the CPU or the motor control logic can be used to generate the phase pulses for the stepper motors. The controller consists of two central counters from which the control pins are derived. The central counters have several registers (see Table 44) used to configure the cycle period, the phase, the duty cycle, and counter granularity.

There are two motor master counters (0 and 1) with identical features. The period of the master counters are defined by the *MotorMasterClkPeriod[1:0]* and *MotorMasterClkSrc* registers i.e. both master counters are derived from the same *MotorMasterClkSrc*. The *MotorMasterClkSrc* defines the timing pulses used by the master counters to determine the timing period. The *MotorMasterClkSrc* can select clock sources of 1μs,100μs,10ms and *pclk* timing pulses.

The *MotorMasterClkPeriod[1:0]* registers are set to the number of timing pulses required before the timing period re-starts. Each master counter is set to the relevant *MotorMasterClkPeriod* value and counts down a unit each time a timing pulse is received.

The master counters reset to *MotorMasterClkPeriod* value and count down. Once the value hits zero a new value is reloaded from the *MotorMasterClkPeriod[1:0]* registers. This ensures that no master clock glitch is generated when changing the clock period.

Each of the IO pins for the motor controller are derived from the master counters. Each pin has independent configuration registers. The *MotorMasterClkSelect[3:0]* registers define which of the two master counters to use as the source for each motor control pin. The master counter value is compared with the configured *MotorCtrlHigh* and *MotorCtrlLow* registers. If the count is equal to *MotorCtrlHigh* value the motor control is set to 1, if the count is equal to *MotorCtrlLow* value the motor control pin is set to 0.

This allows the phase and duty cycle of the motor control pins to be varied at *pclk* granularity.

The motor control generators can be paused at the end of a clock period by setting the *MotorMasterClock-Enable* register to zero. This allows the CPU to re-configure the motor controller without causing a glitch on the output pins.

13.3 LED CONTROL

LED lifetime and brightness can be improved and power consumption reduced by driving the LEDs with a pulsed rather than a DC signal. The source clock for each of the LED pins is a 7.8kHz (128μs period) clock generated from the 1μs clock pulse from the Timers block. The *LEDDutySelect* registers are used to create a signal with the desired waveform. Unpulsed operation of the LED pins can be achieved by using CPU IO direct control. By default the LED pins are controlled by the LED control logic.
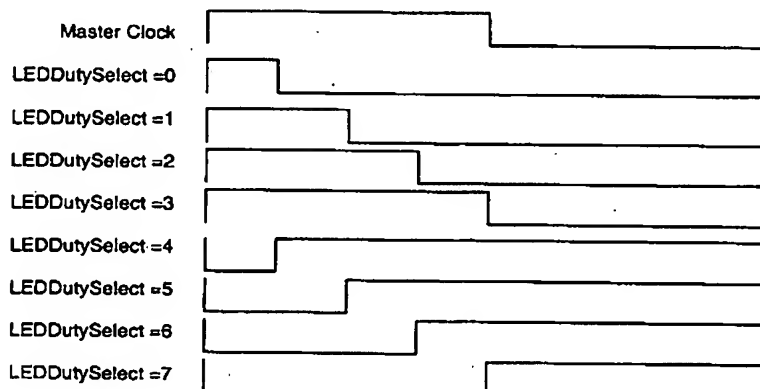
Figure 42. Duty Cycle Select

13.4 LSS INTERFACE VIA GPIO

In some SoPEC system configurations one or more of the LSS interfaces may not be used. Unused LSS interface pins can be reused as general IO pins by configuring the *CpuIOCtrl* register. When a bit in the *CpuIOCtrl* is set the corresponding pin is controlled by the CPU registers, otherwise the pin is controlled by the LSS block. By default the LSS controls the GPIO pins 11 to 8.

13.5 ISI INTERFACE VIA GPIO

In Multi-SoPEC mode the SCB block (in particular the ISI sub-block) requires direct access to and from the *gpio[12]* and *gpio[13]* pins. Control of the ISI interface pins is determined by the *CpuIOCtrl* register.

When a bit in the *CpuIOCtrl* is set the corresponding pin is controlled by the CPU registers, otherwise the pin is controlled by the ISI block directly. By default the pins are directly controlled by the ISI block.

In single SoPEC systems the pins can be re-used by the GPIO.

13.6 CPU GPIO CONTROL

The CPU can assume direct control of any (or all) of the IO pins individually. On a per pin basis the CPU can turn on direct access to the pin by setting the *CpuIOCtrl* register. Once set the IO pin assumes the direction specified by the *CpuIODirection* register. When in output mode the value in register *CpuIOOut*

will be directly reflected to the output driver. When in input mode the status of the input pin can be read in either the direct version or a de-glitched form, by reading *CpuIOIn* and *CpuIOInDeglitch* respectively. When writing to the *CpuIOOut* register the top bits of the register (bits 29 to 16) are used to filter access to the lower bits (13 to 0).

13.7 · PROGRAMMABLE DE-GLITCHING LOGIC

Each IO pin can be filtered through a de-glitching logic circuit. The circuit can be configured to sample the IO pin for a predetermined time before concluding that a pin is in a particular state. The exact sampling length is configurable, but each GPIO pin must use one of two possible configured values (selected by *DeGlitchSelect*). The sampling length is the same for both high and low states. The *DeGlitchCount* is programmed to the number of system time units that a state must be valid for before the state is passed on. The time units are selected by *DeGlitchClkSel* and can be one of 1μs, 100μs, 10ms and *pclk* pulses.

For example if *DeGlitchCount* is set to 10 and *DeGlitchClkSel* set to 3, then an input pin (one of *gpio[13 to 0]*) must consistently retain its value for 10 system clock cycles (*pclk*) before the input state will be propagated from *CpuIOIn* to *CpuIOInDeglitch*.

13.8 INTERRUPT GENERATION

Any of the GPIO pins can generate an interrupt from the raw or deglitched version of the input pin. There are 14 possible interrupt sources from the GPIO to the interrupt controller, one interrupt per input pin. The *InterruptSrcSelect* register determines whether the raw input or the deglitched version is used as the interrupt source.

The interrupt type, masking and priority can be programmed in the interrupt controller.

13.9 FREQUENCY ANALYSER

The frequency analyser measures the duration between successive positive edges on an input pin and reports the last period measured (*FreqAnaLastPeriod*) and a running average period (*FreqAnaAverage*).

The running average is updated each time a new positive edge is detected and is calculated by
$FreqAnaAverage = (FreqAnaAverage / 8) * 7 + FreqAnaLastPeriod / 8$.

The analyser can be used with any input pin (or its deglitched form), but only one pin at a time can be selected. The pin is selected by the *FreqAnaPinSelect* and its deglitched form can be selected by *FreqAnaPinFormSelect*.

Doc: SoPEC_hardware_design
Version: 2.3

S3 Proprietary Document

29 Nov 2002
Page 140

13.10   IMPLEMENTATION

13.10.1   Definitions of I/O

Table 43. I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	In	Timers block generated timing pulses. 0 - 1 μs pulse 1 - 100 μs pulse 2 - 10 ms pulse
CPU Interface			
cpu_addr[7:2]	6	In	CPU address bus. Only 6 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
gpio_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_gpio_sel	1	In	Block select from the CPU. When *cpu_gpio_sel* is high both *cpu_addr* and *cpu_dataout* are valid
gpio_cpu_rdy	1	Out	Ready signal to the CPU. When *gpio_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the GPIO block and for a read cycle this means the data on *gpio_cpu_data* is valid.
gpio_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
gpio_cpu_debug_valid	1	Out	Debug Data valid on *gpio_cpu_data* bus. Active high
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
IO Pins			
gpio_o[13:0]	14	Out	General purpose IO output to IO driver
gpio_i[13:0]	14	In	General purpose IO input from IO receiver
gpio_e[13:0]	14	Out	General purpose IO output control. Active high driving
GPIO to LSS			
lss_gpio_do[1:0]	2	In	LSS bus data output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
gpio_lss_di[1:0]	2	Out	LSS bus data input Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	In	LSS bus data output enable, active high Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	In	LSS bus clock output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
GPIO to ISI			

Table 43. I/O definition

Port name	Pins	I/O	Description
gpio_isi_din[1:0]	2	Out	Input data from IO receivers to ISI.
isi_gpio_dout[1:0]	2	In	Data output from ISI to IO drivers
isi_gpio_e[1:0]	2	In	GPIO ISI pins output enable (active high) from ISI Interface
Interrupts			
gpio_icu_irq[13:0]	14	Out	GPIO pin interrupts
Debug			
debug_data_out[16:3]	14	In	Output debug data to be muxed on to the GPIO pins
debug_cntrl[16:3]	14	In	Control signal for each GPIO bound debug data line indicating whether or not the debug data should be selected by the pin mux

13.10.2 Configuration registers

The configuration registers in the GPIO are programmed via the CPU interface. Refer to section 11.4.3 on page 70 for a description of the protocol and timing diagrams for reading and writing registers in the GPIO. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the GPIO. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *gpio_pcu_data*. Table 44 lists the configuration registers in the GPIO block

Table 44. GPIO Register Definition

Address GPIO_base +	Register	#bits	Reset	Description
CPU IO Control				
0x00	CpuIOCtrl	14	0x0000	Indicates whether each IO pin is directly controlled by the CPU or not 0 - Default Control 1 - CPU Control
0x04	CpuIOUserModeMask	14	0x0000	User Mode Access Mask to CPU GPIO control register. When 1 user access is enabled. One bit per gpio pin. Enables access to *CpuIODirection, CpuIOOut, CpuIOIn* and *CpuIOInDeglitch* in user mode if *CpuIOCtrl* allows CPU access.
0x08	CpuIOSuperModeMask	14	0x3FFF	Supervisor Mode Access Mask to CPU GPIO control register. When 1 supervisor access is enabled. One bit per gpio pin. Enables access to *CpuIODirection, CpuIOOut, CpuIOIn* and *CpuIOInDeglitch* in supervisor mode if *CpuIOCtrl* allows CPU access.
0x0C	CpuIODirection	14	0x0000	Indicates the direction of each IO pin, when controlled by the CPU 0 - Indicates Input Mode 1 - Indicates Output Mode

Table 44. GPIO Register Definition

Address GPIO base+	Register	#bits	Reset	Description
0x10	CpuIOOut	30	0x0000_0000	Value used to drive output pin in CPU direct mode. bits13:0 - Value to drive on output GPIO pins bits 15:14 - Reserved, (Read as zero always) bits 29:16 - Write enable mask for bits13:0, 0 enables write, 1 masks the write. (Read as zero always)
0x14	CpuIOIn	14	External pin value	Value received on each input pin regardless of mode. **Read Only register.**
0x18	CpuIOInDeglitch	14	0x0000	Deglitched version of *CpuIOIn* register. Note that after reset this register will reflect the external pin values 256 *pclk* cycles after they have stabilized. **Read Only register.**
Deglitch control				
0x20-024	DeGlitchCount[1:0]	2x8	0xFF	De-glitch circuit sample count in *DeGlitchClkSrc* selected units for pins *gpio[13:0]*
0x28-2C	DeGlitchClkSrc[1:0]	2x2	0x3	Specifies the unit use of the GPIO deglitch circuits: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x30	DeGlitchSelect	14	0x000	Specifies which deglitch count (*DeGlitchCount*) and unit select (*DeGlitchClkSrc*) should be used to deglitch each GPIO pin 0 - Specifies *DeGlitchCount[0]* and *DeGlitchClkSrc[0]* 1 - Specifies *DeGlitchCount[1]* and *DeGlitchClkSrc[1]*
Motor Control				
0x34	MotorCtrlUserModeEnable	1	0x0	User Mode Access enable to Motor control configuration registers. When 1 user access is enabled. Enables user access to *MotorMasterClkPeriod*, *MotorMasterClkSrc*, *MotorDutySelect*, *MotorPhaseSelect*, *MotorMasterClockEnable* and *MotorMasterClkSelect* registers
0x38 to 0x3C	MotorMasterClkPeriod[1:0]	2x16	0x0000	Specifies the motor controller master clock periods in *MotorMasterClkSrc* selected units
0x40	MotorMasterClkSrc	2	0x0	Specifies the unit use by the motor controller master clock generator: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x44 to 0x50	MotorCtrlHigh[3:0]	4x16	0x0000	Specifies the low to high transition point in the clock period for each motor control pin.
0x54 to 0x60	MotorCtrlLow[3:0]	4x16	0xFFFF	Specifies the high to low transition point in the clock period for each motor control pin.

Table 44. GPIO Register Definition

Address GPIO_base	Register	#bits	Reset	Description
0x64 to 0x70	MotorMasterClkSelect[3:0]	4x1	0x0	Specifies which motor master clock should be used as a pin generator source 0 - Clock derived from *MotorMasterClockPeriod[0]* 1 -Clock derived from *MotorMasterClockPeriod[1]*
0x74	MotorMasterClockEnable	2	0x0	Enable the motor master clock counter. When 1 count is enabled Bit 0 - Enable motor master clock 0 Bit 1 - Enable motor master clock 1
LED control				
0x78	LEDCtrlUserModeEnable	4	0x0	User Mode Access enable to LED control configuration registers. When 1 user access is enabled. One bit per *LEDDutySelect* select register.
0x7C to 0x88	LEDDutySelect[3:0]	4x3	0x0	Specifies the duty cycle for each LED pin.See Figure 42 for encoding details. The *LEDDutySelect[3:0]* registers determine the duty cycle of the *gpio[7:4]* pins
Frequency Analyser				
0x8C	FreqAnaPinSelect	4	0x00	Selects which GPIO input should be used for the frequency analyses.
0x90	FreqAnaPinFormSelect	1	0x0	Selects if the frequency analyser should use the raw input or the deglitched form. 0 - Deglitched form of input pin 1 - Raw form of input pin
0x94	FreqAnaLastPeriod	16	0x0000	Frequency Analyser last period of selected input pin.
0x98	FreqAnaAverage	16	0x0000	Frequency Analyser average period of selected input pin.
0x9C	FreqAnaCountInc	20	0x00000	Frequency Analyser counter increment amount. For each clock cycle no edge is detected on the selected input pin the accumlator is incremented by this amount.
Miscellaneous				
0xA0	InterruptSrcSelect	14	0x000	Interrupt source select.1 bit per GPIO pin. Determines whether the interrupt source is direct form the input pin or the deglitched version 1 - Input pin direct 0 - Deglitched input pin
0xA4	DebugSelect	6	0x00	Debug address select. Indicates the address of the register to report on the *gpio_cpu_data* bus when it is not otherwise being used.
0xA8-0xAC	MotorMasterCount	2x16	0x0000	Motor master clock counter values. Bus 0 - Master clock count 0 Bus 1 - Master clock count 1 **Read Only registers**

13.10.2.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the GPIO will issue a bus error by asserting the *gpio_cpu_berr* signal.

Access to the *CpuIODirection, CpuIOOut, CpuIOIn* and *CpuIOInDeglitch* is filtered by the *CpuIOUserModeMask* and *CpuIOSuperModeMask* registers. Each bit masks access to the corresponding bits in the *CpuIO** registers for each mode, with *CpuIOUserModeMask* filtering user data mode access and *CpuIO-SuperModeMask* filtering supervisor data mode access.

The addition of the *CpuIOSuperModeMask* register helps prevent potential conflicts between user and supervisor code read modify write operations. For example a conflict could exist if the user code is interrupted during a read modify write operation by a supervisor ISR which also modifies the *CpuIO** registers.

An attempt to write to a disabled bit in user or supervisor mode will be ignored, and an attempt to read a disabled bit returns zero. If there are no user mode enabled bits then access is not allowed in user mode and a bus error will result. Similarly for supervisor mode.

When writing to the *CpuIOOut* register, bits 29 to 16 are used to mask the write to the *CpuIOOut[13:0]*. If the mask bit is zero the write is active to corresponding *CpuIOOut* pin, otherwise the write to that pin is ignored.

The pseudocode for determining access to the *CpuIODirection* register is shown below. Similar code could be shown for the *CpuIOOut, CpuIOIn* and *CpuIOInDeglitch* registers.

```
if (cpu_acode == SUPERVISOR_DATA_MODE) then
    // supervisor mode
    if (CpuIOSuperModeMask[13:0] == 0 ) then
        // access is denied, and bus error
        gpio_cpu_berr = 1
    elsif (cpu_rwn == 1) then
        // read mode
        gpio_cpu_data[13:0] = ( CpuIOOut[13:0] & CpuIOSuperModeMask[13:0])
    else
        // write mode,filtered by mask
        mask[13:0]      = ~(cpu_dataout[29:16]) & CpuIOSuperModeMask[13:0]
        CpuIOOut[13:0] = (( cpu_dataout[13:0] &    mask[13:0] ) |
                          ( CpuIOOut[13:0]        & ~(mask[13:0]))))
elsif (cpu_acode == USER_DATA_MODE) then
    // user datamode
    if (CpuIOUserModeMask[13:0] == 0 ) then
        // access is denied, and bus error
        gpio_cpu_berr = 1
    elsif (cpu_rwn == 1) then
        // read mode, filtered by mask
        gpio_cpu_data  = ( CpuIOOut[13:0] & CpuIOUserModeMask[13:0])
    else
        // write mode,filtered by mask
        mask[13:0]      = ~(cpu_dataout[29:16]) & CpuIOUserModeMask[13:0]
        CpuIOOut[13:0] = (( cpu_dataout[13:0] &    mask[13:0] ) |
                          ( CpuIOOut[13:0]        & ~(mask[13:0]))))
else
    // access is denied, bus error
    gpio_cpu_berr = 1
```

Table 45 details the access modes allowed for registers in the GPIO block. In supervisor mode all registers are accessible. In user mode forbidden accesses will result in a bus error (*gpio_cpu_berr* asserted).

Table 45. GPIO supervisor and user access modes

Register Address	Registers	Access Permitted
0x00	CpuIOCtrl	Supervisor data mode only
0x04	CpuIOUserModeMask	Supervisor data mode only
0x08	CpuIOSuperModeMask	Supervisor data mode only
0x0C	CpuIODirection	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x10	CpuIOOut	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x14	CpuIOIn	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x18	CpuIOInDeglitch	CpuIOUserModeMask and CpuIOSuperModeMask filtered
0x20-024	DeGlitchCount[1:0]	Supervisor data mode only
0x28-2C	DeGlitchClkSrc[1:0]	Supervisor data mode only
0x30	DeGlitchSelect	Supervisor data mode only
0x34	MotorCtrlUserModeEnable	Supervisor data mode only
0x38 to 0x3C	MotorMasterClkPeriod[1:0]	MotorCtrlUserModeEnable enabled
0x40	MotorMasterClkSrc	MotorCtrlUserModeEnable enabled
0x44 to 0x50	MotorCtrlHigh[3:0]	MotorCtrlUserModeEnable enabled
0x54 to 0x60	MotorCtrlLow[3:0]	MotorCtrlUserModeEnable enabled
0x64 to 0x70	MotorMasterClkSelect[3:0]	MotorCtrlUserModeEnable enabled
0x74	MotorMasterClockEnable	MotorCtrlUserModeEnable enabled
0x78	LEDCtrlUserModeEnable	Supervisor data mode only
0x80	LEDDutySelect[0]	LEDCtrlUserModeEnable[0] enabled
0x84	LEDDutySelect[1]	LEDCtrlUserModeEnable[1] enabled
0x74	LEDDutySelect[2]	LEDCtrlUserModeEnable[2] enabled
0x88	LEDDutySelect[3]	LEDCtrlUserModeEnable[3] enabled
0x8C	FreqAnaPinSelect	Supervisor data mode only
0x90	FreqAnaPinFormSelect	Supervisor data mode only
0x94	FreqAnaLastPeriod	Supervisor data mode only
0x98	FreqAnaAverage	Supervisor data mode only
0x9C	FreqAnaCountInc	Supervisor data mode only
0xA0	InterruptSrcSelect	Supervisor data mode only
0xA4	DebugSelect	Supervisor data mode only
0xA8-0xAC	MotorMasterCount	Supervisor data mode only

13.10.3 GPIO partition
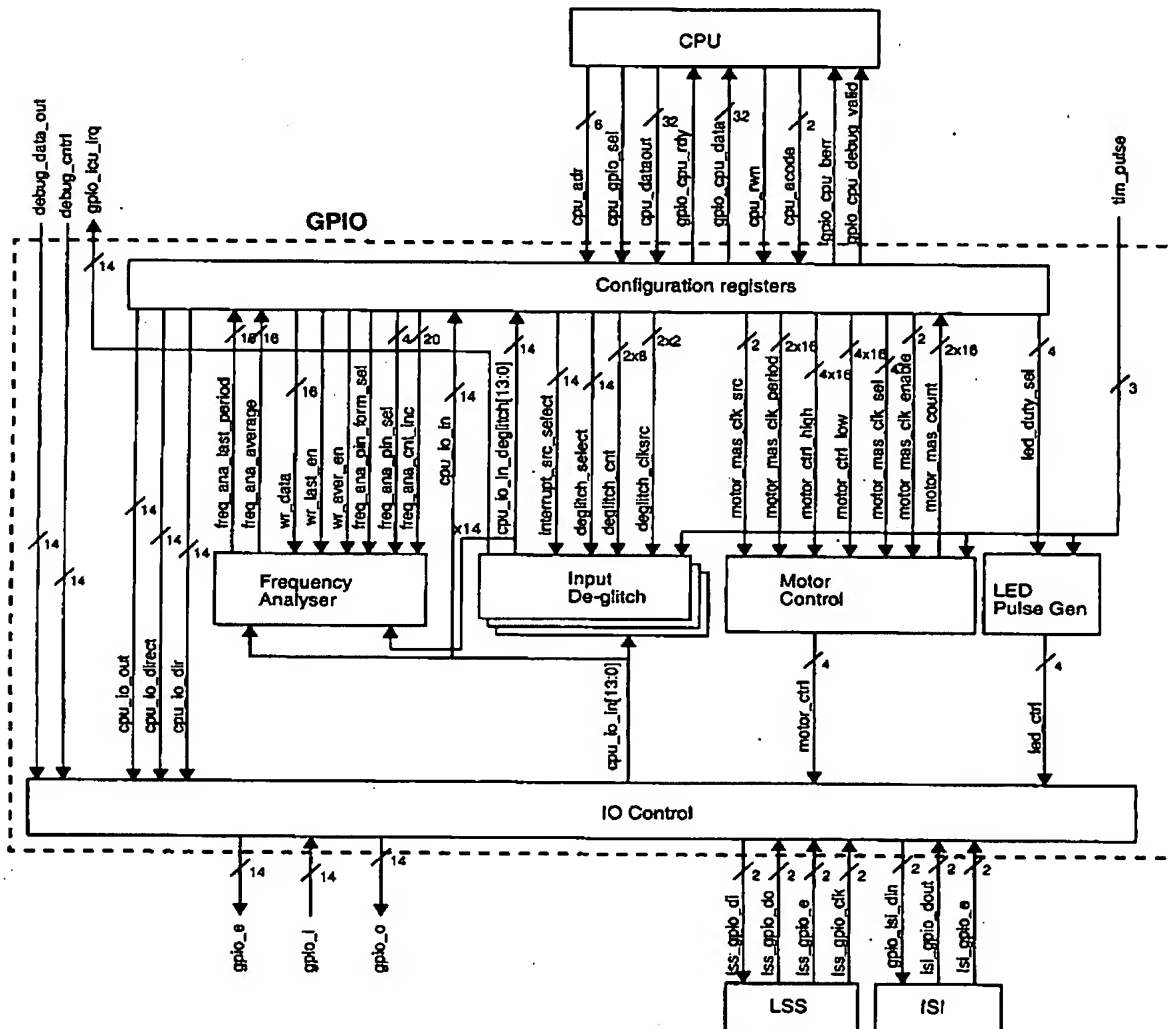

Figure 43. GPIO partition

13.10.4 IO control

The IO control block connects the IO pin drivers to internal signalling based on configured setup registers and debug control signals.

The motor, LED pins, ISI and LSS control logic:

```
// motor and led pins
for (i=0; i<14 ; i++) {
   if (debug_cntrl[i] == 1) then
      gpio_e[i]    = 1
      gpio_o[i]    = debug_data_out[i]
```

```
                  cpu_io_in[i] = gpio_i[i]
           if (cpu_io_ctrl[i] == 1) then
              gpio_e[i]    = cpu_io_dir[i]
              gpio_o[i]    = cpu_io_out[i]
              cpu_io_in[i] = gpio_i[i]
           else
              // default control
              if ( i < 4 ) then // motor control pins
                 gpio_e[i]    = 1
                 gpio_o[i]    = motor_ctrl[i]
                 cpu_io_in[i] = gpio_i[i]
              elsif ( i < 8 ) then // LED pins
                 gpio_e[i]    = 1
                 gpio_o[i]    = led_ctrl[i]
                 cpu_io_in[i] = gpio_i[i]
              elsif (i < 10)  then // LSS interface clock pins
                 gpio_e[i]    = 1
                 gpio_o[i]    = lss_gpio_clk[i-8]
                 cpu_io_in[i] = gpio_i[i]
              elsif (i < 12)  then // LSS interface data pins
                 gpio_e[i]    = lss_gpio_e[i-10]
                 gpio_o[i]    = lss_gpio_do[i-10]
                 lss_gpio_di[i-10] = gpio_i[i]
              else                 // ISI interface pins
                 gpio_e[i]    = isi_gpio_e[i-12]
                 gpio_o[i]    = isi_gpio_dout[i-12]
                 isi_gpio_din[i-12] = gpio_i[i]
           }
```

13.10.5 LED pulse generator

The pulse generator logic consists of a 7-bit counter that is incremented on a 1μs pulse from the timers block (*tim_pulse[0]*). The LED control signal is generated from comparing the count value with the configured duty cycle for the LED (*led_duty_sel*).

The logic is given by:
```
for (i=0 i<4 ;i++) { // for each LED pin
   // period divided into 8 segments
   period_div8 = cnt[6:4];
   if (period_div8 <= led_duty_sel[i]) then
      led_ctrl[i] = 1
   else
      led_ctrl[i] = 0
   // in higher half invert the led control
   if (cnt[6] == 1) then
      led_ctrl[i] = ~ led_ctrl[i]
   }
// update the counter every 1us pulse
if (tim_pulse[0] == 1) then
   cnt ++ .
```

13.10.6 Motor control

The motor controller consists of 2 counters, and 4 phase generator logic blocks, one per motor control pin. The counters decrement each time a timing pulse (*cnt_en*) is received. The counters start the configured clock period value (*motor_mas_clk_period*) and decrement to zero. If the counters are enabled (via *motor_mas_clk_enable*), the counters will automatically restart at the configured clock period value, otherwise they will wait until the counters are re-enabled.

The timing pulse period is one of *pclk*, 1µs, 100µs, 1ms depending on the *motor_mas_clk_sel* signal. The counters are used to derive the phase and duty cycle of the of each motor control pin.

```
// decrement logic
if (cnt_en == 1) then
    if ((mas_cnt == 0) AND (motor_mas_clk_enable == 1)) then
        mas_cnt = motor_mas_clk_period[15:0]
    elsif ((mas_cnt == 0) AND (motor_mas_clk_enable == 0)) then
        mas_cnt = 0
    else
        mas_cnt --
else    // hold the value
    mas_cnt = mas_cnt
```
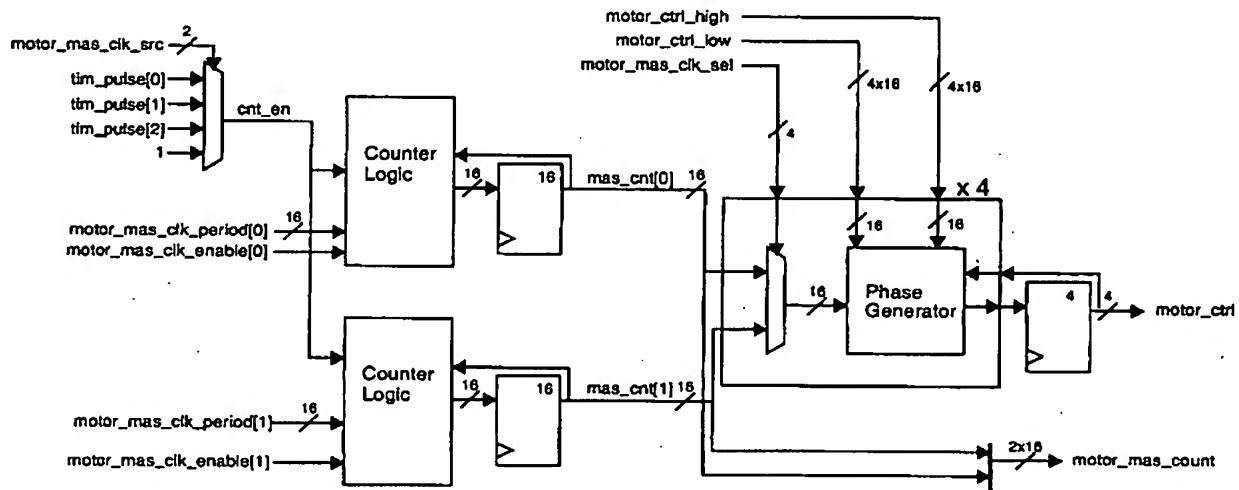

Figure 44. Motor control RTL diagram

The phase generator block generates the motor control logic based on the selected clock generator (*motor_mas_clk_sel*) the motor control high transition point (*motor_ctrl_high*) and the motor control low transition point (*motor_ctrl_low*). There are 4 instances one per motor control pin.

The logic is given by:

```
// select the input counter to use
if (motor_mas_clk_sel == 1) then
    count   = mas_cnt[1]
else
    count   = mas_cnt[0]
// Generate the phase and duty cycle
if ((motor_ctrl == 1 ) AND (count == motor_ctrl_low)) then
    motor_ctrl = 0
elsif ((motor_ctrl == 0) AND (count == motor_ctrl_high)) then
    motor_ctrl = 1
else
    motor_ctrl = motor_ctrl // remain the same
```

13.10.7 Input deglitch

The input deglitch logic rejects input states of duration less than the configured number of time units (*deglitch_cnt*), input states of greater duration are reflected on the output *cpu_io_in_deglitch*. The time units used (either *pclk*, 1μs, 100μs, 1ms) by the deglitch circuit is selected by the *deglitch_clk_src* bus.

There are 2 possible sets of *deglitch_cnt* and *deglitch_clk_src* that can be used to deglitch the input pins. The values used are selected by the *deglitch_sel* signal.

Each input pin can be used to generate an interrupt. The interrupt can be generated from the raw input signal or a deglitched version of the input. The interrupt source is selected by the *interrupt_src_select* signal.

The counter logic is given by

```
if ( cpu_io_in != cpu_io_in_delay) then
    cnt         = deglitch_cnt
    output_en = 0
elsif (cnt == 0 ) then
    cnt         = cnt
    output_en = 1
elsif (cnt_en == 1) then
    cnt --
    output_en = 0
```
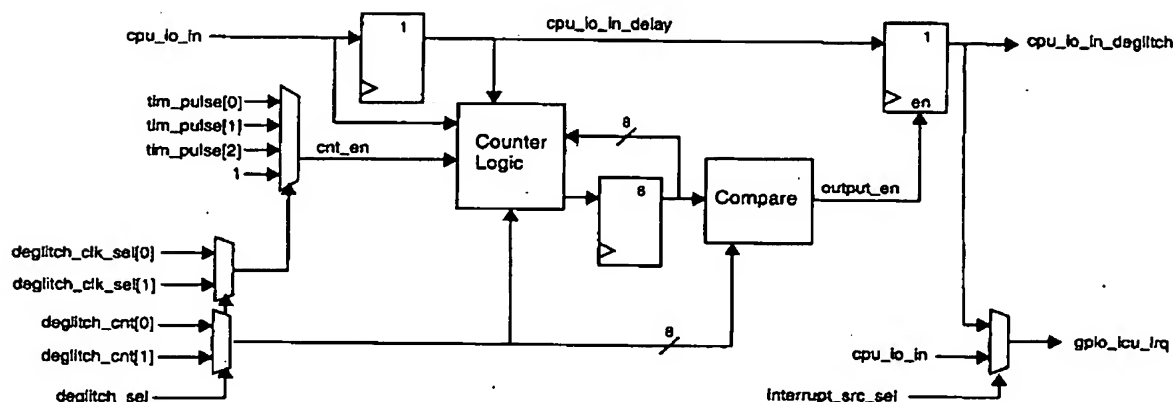

Figure 45. Input de-glitch RTL diagram

13.10.8 Frequency Analyser

The frequency analyzer block monitors a selected input pin (selected by *FreqAnaPinSelect* and *FreqAnaPinFormSel*) and detects positive edges. Between successive positive edges detected on the input pin it increments a counter by a programmed amount (*FreqAnaCountInc*) on each clock cycle. When a positive edge is detected the *FreqAnaLastPeriod* register is updated with the top 16 bits of the counter and the counter is reset. The frequency analyser also maintains a running average of the *FreqAnaLastPeriod* register. Each time a positve edge is detected on the input pin the *FreqAnaAverage* register is updated with the new calculated *FreqAnaLastPeriod*. The average is calculated as 7/8 the current value plus 1/8 of the new value. Both the *FreqAnaLastPeriod* and *FreqAnaAverage* registers can be written to by the CPU.

The pseudocode is given by

```
if ((pin == 1) AND pin_delay ==0 ))then      // positive edge detected
    freq_ana_lastperiod = count[31:16]
    freq_ana_average    = freq_ana_average - freq_ana_average/8 + freq_ana_lastperiod/8
```

```
        count = 0
    else
        count = count + freq_ana_count_inc
    // implement the configuration register write
    if (wr_last_en == 1) then
        freq_ana_lastperiod = wr_data
    elsif (wr_average_en == 1 ) then
        freq_ana_average = wr_data
```
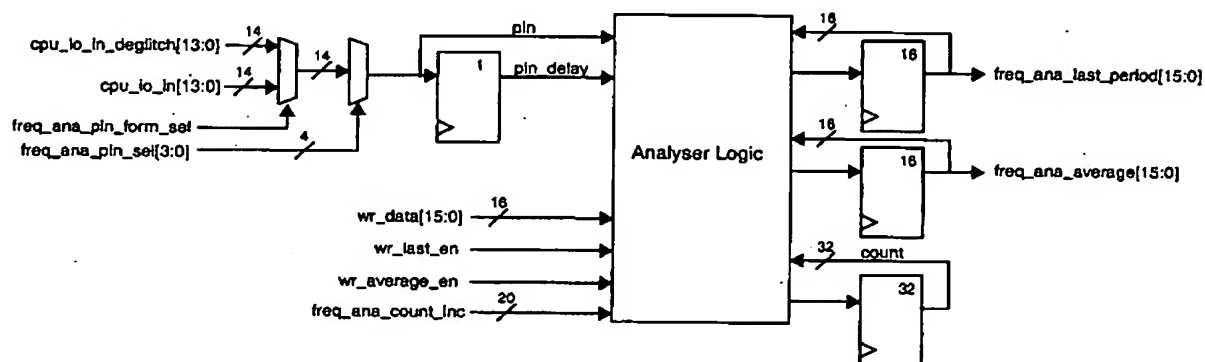

Figure 46. Frequency analyser RTL diagram

14 Interrupt Controller Unit (ICU)

The interrupt controller accepts up to N input interrupt sources, determines their priority, arbitrates based on the highest priority and generates an interrupt request to the CPU. The ICU complies with the interrupt acknowledge protocol of the CPU. Once the CPU accepts an interrupt (i.e. processing of its service routine begins) the interrupt controller will assert the next arbitrated interrupt if one is pending.

Each interrupt source has a fixed vector number N, and an associated configuration register, *IntReg[N]*. The format of the *IntReg[N]* register is shown in Table 46 below.

Table 46. IntReg[N] register format

Field	bit(s)	Description
Priority	7:0	Interrupt priority
Type	9:8	Determines the triggering conditions for the interrupt 00 - Positive edge 10 - Negative edge 01 - Positive level 11 - Negative level
Mask	10	Mask bit. 1 - Interrupts from this source are enabled, 0 - Interrupts from this source are disabled. Note that there may be additional masks in operation at the source of the interrupt.
Reserved	31:11	Reserved. Write as 0.

Once an interrupt is received the interrupt controller determines the priority and maps the programmed priority to the available CPU priority levels, and then issues an interrupt to the CPU. The mapping of programmed priority to native interrupt levels will be fixed, and is dependent on CPU choice.

For example for the LEON CPU there are 15 levels available which would allow 16 sub-priorities per level (as each level is in itself a priority). In this case priorities 255-240 map to level 15, 240-224 to level 14 and so on, with priorities 15-0 corresponding to level 0. Level 0 is no interrupt. Level 15 is the highest interrupt level.

14.1 INTERRUPT PREEMPTION

There are two types of pre-emption possible: standard LEON pre-emption and SoPEC pending pre-emption. With standard LEON pre-emption an interrupt can only be pre-empted by an interrupt with a higher priority level. If an interrupt with the same priority level (1 to 15) as the interrupt being serviced becomes pending then it is not acknowledged until the current service routine has completed. The SoPEC pending pre-emption is an extension of the standard LEON scheme which is made possible by the programmable priority levels in the *IntReg[N]* register.

Interrupts with a higher sub-priority will pre-empt interrupts with a lower sub-priority but the same priority level mapping, if the interrupt has not been acknowledged by the CPU i.e. it is still pending. If an interrupt with a higher sub-priority arrives while an interrupt with a lower sub-priority at the same level is being serviced then it will not be serviced until the lower sub-priority service routine has completed.

Thus when pre-emption is required, interrupts should be programmed to different levels as interrupt priorities of the same level have no guaranteed servicing order.

The interrupt is directly acknowledged by the CPU and the ICU automatically clears the pending bit of acknowledged interrupts.

All interrupt controller registers are only accessible in supervisor data mode. If the user code wishes to mask an interrupt it must request this from the supervisor and the supervisor software will resolve user access levels.

14.2 INTERRUPT SOURCES

The mapping of interrupt sources to interrupt vectors (and therefore *IntReg[N]* registers) is shown in Table 47 below. Please refer to the appropriate section of this specification for more details of the interrupt sources.

Table 47. Interrupt sources vector table

Vector	Source	Description
0	Timers	WatchDog Timer Update request
1	Timers	Generic Timer 1 interrupt
2	Timers	Generic Timer 2 interrupt
3	Timers	Generic Timer 3 interrupt
4 - 17	GPIO	GPIO general interrupt, source pin 0 -13
18	MMU	MMU Security violation
19	SCB	USB interrupt
20	SCB	ISI interrupt
21	SCB	DMA interrupt
22	LSS	LSS interrupt, LSS interface 0 interrupt request
23	LSS	LSS interrupt, LSS interface 1 interrupt request
24	PCU	PEP Sub-system interrupt- CDU finished band
25	PCU	PEP Sub-system interrupt- CDU error
26	PCU	PEP Sub-system interrupt- LBD finished band
27	PCU	PEP Sub-system interrupt- TE finished band
28	PCU	PEP Sub-system interrupt- PCU finished band
29	PCU	PEP Sub-system interrupt- PCU invalid address interrupt
30	PCU	PEP Sub-system interrupt- PHI Buffer underrun
31	PCU	PEP Sub-system interrupt- PHI Page finished
32	PCU	PEP Sub-system interrupt- PHI Print ready
33	PHI	PEP Sub-system interrupt- PHI Line Sync interrupt

14.3   IMPLEMENTATION

14.3.1   Definitions of I/O

Table 48. Interrupt Controller Unit I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface			
cpu_adr[7:2]	6	In	CPU address bus. Only 6 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
icu_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_icu_sel	1	In	Block select from the CPU. When *cpu_icu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
icu_cpu_rdy	1	Out	Ready signal to the CPU. When *icu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the ICU block and for a read cycle this means the data on *icu_cpu_data* is valid.
icu_cpu_ilevel[3:0]	4	Out	Indicates the priority level of the current active interrupt.
cpu_iack	1	Out	Interrupt request acknowledge from the LEON core.
cpu_icu_ilevel[3:0]	4	In	Interrupt acknowledged level from the LEON core
icu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
icu_cpu_debug_valid	1	Out	Debug Data valid on *icu_cpu_data* bus. Active high
Interrupts			
tim_icu_wd_irq	1	In	Watchdog timer interrupt signal from the Timers block
tim_icu_irq[2:0]	3	In	Generic timer interrupt signals from the Timers block
gpio_icu_irq[13:0]	14	In	GPIO pin interrupts
mmu_icu_irq	1	In	Memory Management Unit interrupt
usb_icu_irq	1	In	USB interrupt from the SCB
isi_icu_irq	1	In	ISI interrupt from the SCB
dma_icu_irq	1	In	DMA interrupt from the SCB
lss_icu_irq[1:0]	2	In	LSS interface interrupt request
cdu_finishedband	1	In	Finished band interrupt request from the CDU
cdu_icu_jpegerror	1	In	JPEG error interrupt from the CDU
lbd_finishedband	1	In	Finished band interrupt request from the LBD
te_finishedband	1	In	Finished band interrupt request from the TE
pcu_finishedband	1	In	Finished band interrupt request from the PCU
pcu_icu_address_invalid	1	In	Invalid address interrupt request from the PCU

Table 48. Interrupt Controller Unit I/O definition

Port name	# Pins	I/O	Description
phi_icu_underrun	1	In	Buffer underrun interrupt request from the PHI
phi_icu_page_finish	1	In	Page finished interrupt request from the PHI
phi_icu_print_rdy	1	In	Print ready interrupt request from the PHI
phi_icu_linesync_int	1	In	Line sync interrupt request from the PHI

14.3.2   Configuration registers

The configuration registers in the ICU are programmed via the CPU interface. Refer to section 11.4 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the ICU. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the ICU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *icu_pcu_data*. Table 49 lists the configuration registers in the ICU block.

The ICU block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *icu_cpu_berr* being asserted.

Table 49. ICU Register Map

| Address ICU base	Register	#bits	Reset	Description
0x00 - 0x84	IntReg[33:0]	34x11	0x000	Interrupt vector configuration register
0x88-0x8C	IntClear[1:0]	2x32	0x0000_0000	Interrupt pending clear register. If written with a one it clears corresponding interrupt IntClear[0] - Interrupts sources 31 to 0 IntClear[1] - Interrupts source 33 to 32
0x90-0x94	IntPending[1:0]	2x32	0x0000_0000	Interrupt pending register. (Read Only) IntPending[0] - Interrupts sources 31 to 0 IntPending[1] - Interrupts source 33 to 32
0x98	IntSource	6	0x00	Indicates the interrupt source of the current winning active interrupt. (Read Only)
0x9C	DebugSelect	6	0x00	Debug address select. Indicates the address of the register to report on the *icu_cpu_data* bus when it is not otherwise being used.

14.3.3 ICU partition
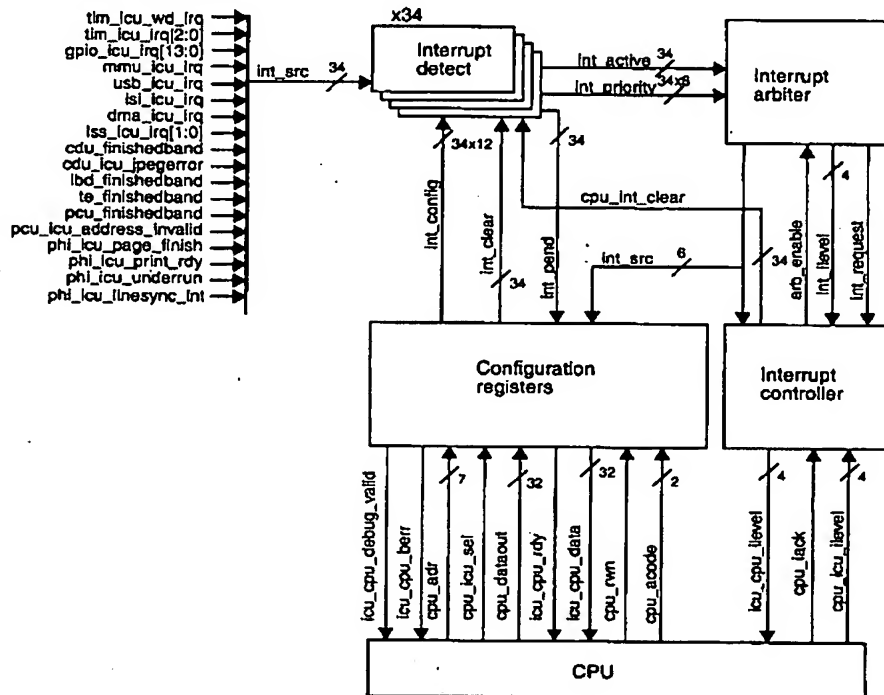
Figure 47. ICU partition

14.3.4 Interrupt detect

The ICU contains multiple instances of the interrupt detect block, one per interrupt source. The interrupt detect block examines the interrupt source signal, and determines whether it should generate request pending (*int_pend*) based on the configured interrupt type and the interrupt source conditions. If the interrupt is not masked the interrupt will be reflected to the interrupt arbiter via the *int_active* signal. Once an interrupt is pending it remains pending until the interrupt is accepted by the CPU or it is level sensitive and gets removed. Masking a pending interrupt has the effect of removing the interrupt from arbitration but the interrupt will still remain pending.

When the CPU accepts the interrupt (using the normal ISR mechanism), the interrupt controller automatically generates an interrupt clear for that interrupt source (*cpu_int_clear*). Alternatively if the interrupt is masked, the CPU can determine pending interrupts by polling the *IntPending* registers. Any active pending interrupts can be cleared by the CPU without using an ISR via the *IntClear* registers.

The logic is shown below:
```
mask          = int_config[10]
type          = int_config[9:8]
int_priority = int_config[7:0]
int_pend     = last_int_pend              // the last pending interrupt
// update the pending FF
if ((int_clear == 1 )OR (cpu_int_clear==1)) then
   int_pend = 0
// test for interrupt condition
if ((type == NEG_LEVEL ) AND (int_src == 0) then
```

```
    int_pend = 1
elsif ((type == POS_LEVEL) AND (int_src == 1)
    int_pend = 1
elsif ((type == NEG_EDGE ) AND (int_src == 1) AND (last_int_src == 0))
    int_pend = 1
elsif ((type == POS_EDGE ) AND (int_src == 0) AND (last_int_src == 1))
    int_pend = 1
else
    int_pend = last_int_src // stay the same as before
// mask the pending bit
if (mask == 1) then
    int_active = int_pend
else
    int_active = 0
// assign the registers
last_int_src  = int_src
last_int_pend = int_pend
```

14.3.5 Interrupt arbiter

The interrupt arbiter logic arbitrates a winning interrupt request from multiple pending requests based on configured priority. It generates the interrupt to the CPU by setting *icu_cpu_ilevel* to a non-zero value. The priority of the interrupt is reflected by the value assigned to *icu_cpu_ilevel*, the higher the value the higher the priority, 15 being the highest. The current winning interrupt and is reported to the CPU via the *IntSrc* register generated in the interrupt arbiter block.

```
// arbitrate based on priority
if (arb_enable == 1 ) then
    // arbitrate with the current winner
    win_int_priority = 0
    int_src          = 0
    int_request      = 0
    for (i=0;i<34;i++) {
        if ( int_active[i] == 1) then {
            if (int_priority[i] > win_int_priority ) then
                win_int_priority = int_priority[i]
                int_src          = i
                int_request      = 1
            }
        }
    } .
    // assign the CPU interrupt level
    int_ilevel = int_priority[int_src][7:4]
}
```

14.3.6 Interrupt controller

The interrupt controller is responsible for generating the interrupt to the CPU, accepting the interrupt acknowledge from the CPU and clearing the interrupt source pending bit.

The exact procedure is CPU dependent, but examples are given for the LEON processor. See section 11.9 on page 98 for a complete description of the interrupt handling procedure.
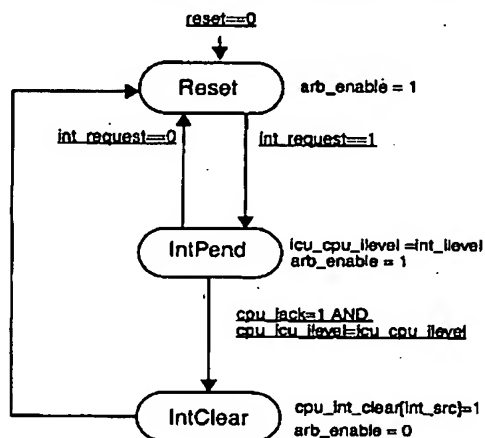
Figure 48. Interrupt controller state diagram

After reset the interrupt controller remains in the *Reset* state until the interrupt arbiter indicates that there is an active interrupt pending (*int_request* equal 1). The state machine goes to the *IntPend* state and signals to the CPU that an interrupt is pending. The machine will remain in the *IntPend* state until the interrupt is acknowledged by the CPU or the pending interrupt condition is removed.

When the interrupt is acknowledged the state machine goes to the *IntClear* state to clear the pending bit of the interrupt source.

On completion the state machine returns to the *Reset* state and again waits for the next pending interrupt.

15 Timers Block (TIM)

The Timers block contains general purpose timers, a watchdog timer and timing pulse generator for use in other sections of SoPEC.

15.1 WATCHDOG TIMER

The watchdog timer is a 32 bit counter value which counts down each time a timing pulse is received. The period of the timing pulse is selected by the *WatchDogUnitSel* register. The value at any time can be read from the *WatchDogTimer* register and the counter can be reset by writing a non-zero value to the register. Should the counter reach 1, a system wide reset will be triggered as if the reset came from a hardware pin.

The watchdog timer can be polled by the CPU and reset each time it gets close to 1, or alternatively a threshold (*WatchDogIntThres*) can be set to trigger an interrupt for the watchdog timer to be serviced by the CPU. This interrupt can be effectively masked by setting the threshold to zero. The watchdog timer can be disabled, without causing a reset, by writing zero to the *WatchDogTimer* register.

15.2 TIMING PULSE GENERATOR

The timing block contains a timing pulse generator clocked by the system clock, used to generate timing pulses of 1μs, 100μs and 10ms. Each pulse is of one system clock duration and is active high, with the pulse period accurate to the system clock frequency.

The timing pulse generator also contains a 64-bit free running counters that can be read or reset by accessing the *FreeRunCount* register.

15.3 GENERIC TIMERS

SoPEC contains 3 programmable generic timing counters, for use by the CPU to time the system. The timers are programmed to a particular value and count down each time a timing pulse is received. If a particular timer decrements to 0, then an interrupt is generated. The counter can be programmed to automatically restart the count, or wait until re-programmed by the CPU. At any time the status of the counter can be read from *GenCntValue*, or can be reset by writing to *GenCntValue* register. The auto-restart is activated by setting the *GenCntAuto* register, when activated the counter restarts at *GenCntStartValue*. A counter can be stopped or started at any time, without affecting the contents of the *GenCntValue* register, by writing a 1 or 0 to the relevent *GenCntEnable* register.

15.4   IMPLEMENTATION

15.4.1   Definitions of I/O

Table 50. Timers block I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
tim_pulse[2:0]	3	Out	Timers block generated timing pulses, each one pclk wide 0 - 1µs pulse 1 - 100 µs pulse 2 - 10ms pulse
CPU Interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for the ICU block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
tim_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_tim_sel	1	In	Block select from the CPU. When *cpu_tim_sel* is high both *cpu_adr* and *cpu_dataout* are valid
tim_cpu_rdy	1	Out	Ready signal to the CPU. When *tim_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the TIM block and for a read cycle this means the data on *tim_cpu_data* is valid.
tim_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
tim_cpu_debug_valid	1	Out	Debug Data valid on *tim_cpu_data* bus. Active high
Miscellaneous			
tim_icu_wd_irq	1	Out	Watchdog timer interrupt signal to the ICU block
tim_icu_irq[2:0]	3	Out	Generic timer interrupt signals to the ICU block
tim_cpr_reset_n	1	Out	Watch dog timer system reset.

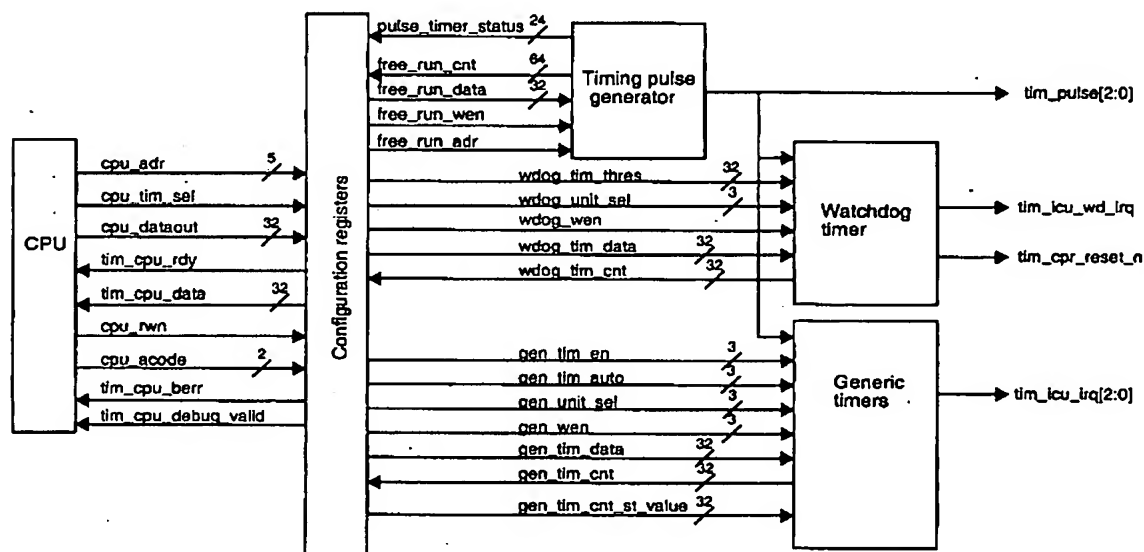## 15.4.2   Timers sub-block partition

Figure 49. Timers sub-block partition diagram

15.4.3   Watchdog timer

The watchdog timer counts down from pre-programmed value, and generates a system wide reset when equal to one. When the counter passes a pre-programmed threshold (*wdog_tim_thres*) value an interrupt is generated (*tim_icu_wd_irq*) requesting the CPU to update the counter. Setting the counter to zero disables the watchdog reset. In supervisor mode the watchdog counter can be written to or read from at any time, in user mode access is denied. Any accesses in user mode will generate a bus error.
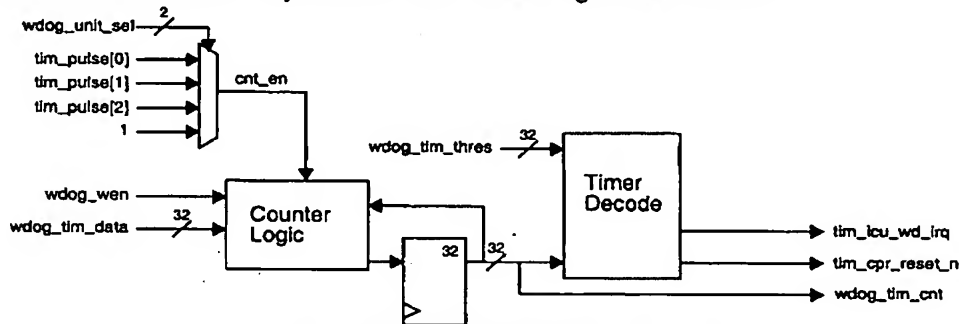

Figure 50. Watchdog timer RTL diagram

The counter logic is given by
```
if (wdog_wen == 1) then
    wdog_tim_cnt = wdog_tim_data      // load new data
elsif ( wdog_tim_cnt == 0) then
    wdog_tim_cnt = wdog_tim_cnt       // count disabled
elsif ( cnt_en == 1 ) then
```

```
            wdog_tim_cnt--
    else
        wdog_tim_cnt = wdog_tim_cnt
```

The timer decode logic is

```
if (( wdog_tim_cnt == wdog_tim_thres) AND (wdog_tim_cnt != 0 )) then
    tim_icu_wd_irq = 1
else
    tim_icu_wd_irq = 0
// reset generator logic
if (wdog_tim_cnt == 1) then
    tim_cpr_reset_n = 0
else
    tim_cpr_reset_n = 1
```

15.4.4 Generic timers

The generic timers block consists of 3 identical counters. A timer is set to a pre-configured value (*GenCnt-StartValue*) and counts down once per selected timing pulse (*gen_unit_sel*). The timer can be enabled or disabled at any time (*gen_tim_en*), when disabled the counter is stopped but not cleared. The timer can be set to automatically restart (*gen_tim_auto*) after it hits zero. In supervisor mode a timer can be written to or read from at any time, in user mode access is determined by the *GenCntUserModeEnable* register settings.
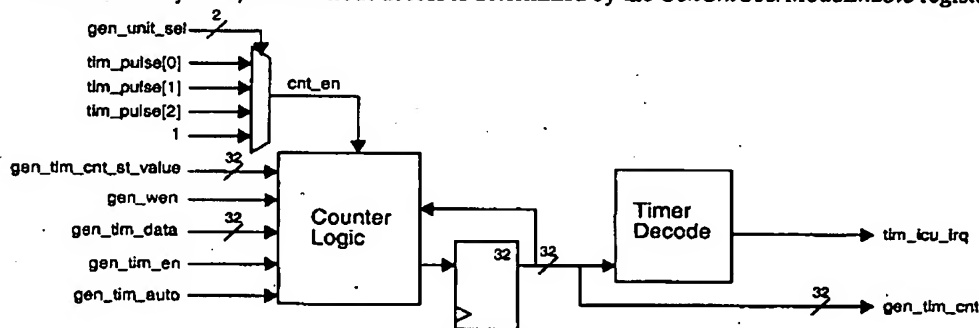
Figure 51. Generic timer RTL diagram

The counter logic is given by

```
if (gen_wen == 1) then
    gen_tim_cnt = gen_tim_data
elsif (( cnt_en == 1 )AND (gen_tim_en == 1 )) then
    if ( gen_tim_cnt == 0) then      // counter may need re-starting
        if (gen_tim_auto == 1) then
            gen_tim_cnt = gen_tim_cnt_st_value
        else
            gen_tim_cnt = gen_tim_cnt
    else
        gen_tim_cnt--
else
    gen_tim_cnt = gen_tim_cnt
```

The decode logic is

```
if (gen_tim_cnt == 1) then
    tim_icu_irq = 1
else
    tim_icu_irq = 0
```

15.4.5   Timing pulse generator

The timing pulse generator contains a general free running 64-bit timer and 3 timing pulse generators producing timing pulses of one cycle duration with a period of 1µs, 100µs and 1ms. In supervisor mode the free running timer register can be written to or read from at any time, in user mode access is denied. The status of each of the 1µs, 100µs and 1ms timer can be read by accessing the *TimerPulseStatus* registers. Any accesses in user mode will result in a bus error. The status of each of the 1µs, 100µs and 1ms timer can be read by accessing the *TimerPulseStatus* register in supervisor mode.
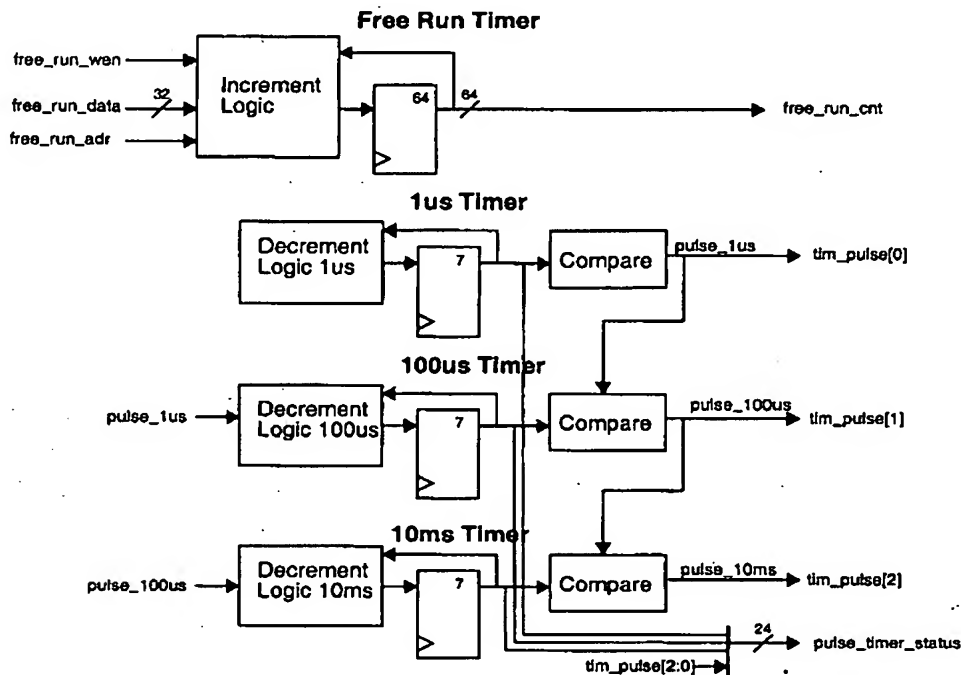
Figure 52. Pulse generator RTL diagram

15.4.5.1 Free Run Timer

The increment logic block increments the timer count on each clock cycle. The counter wraps around to zero and continues incrementing if overflow occurs. When the timing register (*FreeRunCount*) is written to, the configuration registers block will set the *free_run_wen* high for a clock cycle and the value on *free_run_data* will become the new count value, for the 32 bits selected by the *free_run_adr* signal. If *free_run_adr* is 1 the higher 32 bits of the counter will be written to, otherwise the lower 32 bits are written to. It is the responsibility of software to handle these writes in a sensible manner.

The increment logic is given by

```
if (free_run_wen == 1) then
   if (free_run_adr == 1) then
      free_run_cnt[63:32] = free_run_data
   else
      free_run_cnt[31:0]  = free_run_data
else
```

```
free_run_cnt ++
```

15.4.5.2 Pulse Timers

The pulse timer logic generates timing pulses of 1 clock cycle length and period of 1µs, 100µs and 1ms.
The logic for the 1µs timer is given by:

```
// 1us generator
if (pulse_1us_cnt == 0 ) then
   pulse_1us_cnt = 159
   pulse_1us    = 1
else
   pulse_1us_cnt --
   pulse_1us    = 0
```

The logic for 100µs timer is given by:

```
// 100us generator
if ((pulse_100us_cnt == 0 ) AND (pulse_1us == 1)) then
   pulse_100us_cnt = 99
   pulse_100us    = 1
elsif (pulse_1us == 1) then
   pulse_100us_cnt --
   pulse_100us    = 0
else
   pulse_100us_cnt --
   pulse_100us    = 0
```

The logic for the 10ms timer is given by:

```
// 10ms generator
if ((pulse_10ms_cnt == 0 ) AND (pulse_100us == 1)) then
   pulse_10ms_cnt = 99
   pulse_10ms    = 1
elsif (pulse_100us == 1) then
   pulse_10ms_cnt --
   pulse_10ms    = 0
else
   pulse_10ms_cnt --
   pulse_10ms    = 0
```

15.4.6   Configuration registers

The configuration registers in the TIM are programmed via the CPU interface. Refer to section 11.4.3 on
page 70 for a description of the protocol and timing diagrams for reading and writing registers in the TIM.
Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and
writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the TIM.

When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *tim_pcu_data*. Table 51 lists the configuration registers in the TIM block .

Table 51. Timers Register Map

Address TIM base	Register	#bits	Reset	Description
0x00	WatchDogUnitSel	3	0x0	Specifies the units used for the watchdog timer: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x04	WatchDogTimer	32	0xFFFF_FFFF	Specifies the number of units to count before watchdog timer triggers.
0x08	WatchDogIntThres	32	0x0000_0000	Specifies the threshold value below which the watchdog timer issues an interrupt
0x0C-0x10	FreeRunCount[1:0]	2x32	0x0000_0000	Direct access to the free running counter register. Bus 0 - Access to bits 31-0 Bus 1 - Access to bits 63-32
0x14 to 0x1C	GenCntStartValue[2:0]	3x32	0x0000_0000	Generic timer counter start value, number of units to count before event
0x20 to 0x28	GenCntValue[2:0]	3x32	0x0000_0000	Direct access to generic timer counter registers
0x2C to 0x34	GenCntUnitSel[2:0]	3x2	0x0	Generic counter unit select. Selects the timing units used with corresponding counter: 0 - 1 µs pulse 1 - 100 µs pulse 2 - 10 ms pulse 3 - *pclk*
0x38 to 0x40	GenCntAuto[2:0]	3x1	0x0	Generic counter auto re-start select. When high timer automatically restarts, otherwise timer stops.
0x44 to 0x4C	GenCntEnable[2:0]	3x1	0x0	Generic counter enable. 0 - Counter disabled 1 - Counter enabled
0x50	GenCntUserModeEnable	3	0x0	User Mode Access enable to generic timer configuration register. When 1 user access is enabled. Bit 0 - Generic timer 0 Bit 1 - Generic timer 1 Bit 2 - Generic timer 2
0x54	DebugSelect	6	0x00	Debug address select. Indicates the address of the register to report on the *tim_cpu_data* bus when it is not otherwise being used.
Read Only Registers				
0x58	PulseTimerStatus	24	0x00	Current pulse timer values, and pulses 6:0 - 1us timer count 7 - 1us pulse 14:8 - 100us timer count 15 - 100us pulse 22:16- 10ms timer count 23 - 10 ms pulse

15.4.6.1 Supervisor and user mode access

The configuration registers block examines the CPU access type (*cpu_acode* signal) and determines if the access is allowed to that particular register, based on configured user access registers. If an access is not allowed the block will issue a bus error by asserting the *tim_cpu_berr* signal.

The timers block is fully accessible in supervisor data mode, all registers can written to and read from. In user mode access is denied to all registers in the block except for the generic timer configuration registers that are granted user data access. User data access for a generic timer is granted by setting corresponding bit in the *GenCntUserModeEnable* register. This can only be changed in supervisor data mode. If a particular timer is granted user data access then all registers for configuring that timer will be accessible. For example if timer 0 is granted user data access the *GenCntStartValue[0]*, *GenCntUnitSel[0]*, *GenCntAuto[0]*, *GenCntEnable[0]* and *GenCntValue[0]* registers can all be written to and read from without any restriction.

Attempts to access a user data mode disabled timer configuration register will result in a bus error.

Table 52 details the access modes allowed for registers in the TIM block. In supervisor data mode all registers are accessable. All forbidden accesses will result in a bus error (*tim_cpu_berr* asserted).

Table 52. TIM supervisor and user access modes

Register Address	Registers	Access Permission
0x00	WatchDogUnitSel	Supervisor data mode only
0x04	WatchDogTimer	Supervisor data mode only
0x08	WatchDogIntThres	Supervisor data mode only
0x0C-0x10	FreeRunCount	Supervisor data mode only
0x14	GenCntStartValue[0]	GenCntUserModeEnable[0]
0x18	GenCntStartValue[1]	GenCntUserModeEnable[1]
0x1C	GenCntStartValue[2]	GenCntUserModeEnable[2]
0x20	GenCntValue[0]	GenCntUserModeEnable[0]
0x24	GenCntValue[1]	GenCntUserModeEnable[1]
0x28	GenCntValue[2]	GenCntUserModeEnable[2]
0x2C	GenCntUnitSel[0]	GenCntUserModeEnable[0]
0x30	GenCntUnitSel[1]	GenCntUserModeEnable[1]
0x34	GenCntUnitSel[2]	GenCntUserModeEnable[2]
0x38	GenCntAuto[0]	GenCntUserModeEnable[0]
0x3C	GenCntAuto[1]	GenCntUserModeEnable[1]
0x40	GenCntAuto[2]	GenCntUserModeEnable[2]
0x44	GenCntEnable[0]	GenCntUserModeEnable[0]
0x48	GenCntEnable[1]	GenCntUserModeEnable[1]
0x4C	GenCntEnable[2]	GenCntUserModeEnable[2]
0x50	GenCntUserModeEnable	Supervisor data mode only
0x54	DebugSelect	Supervisor data mode only
0x58	PulseTimerStatus	Supervisor data mode only

16 Clocking, Power and Reset (CPR)

The CPR block provides all of the clock, power enable and reset signals to the SoPEC device.

16.1 POWERDOWN MODES

The CPR block is capable of powering down certain sections of the SoPEC device. When a section is powered down (i.e. put in sleep mode) no state is retained, the CPU must re-initialize the section before it can be used again. The exact powerdown mechanism is undefined and is technology dependent.

For the purpose of powerdown the SoPEC device is divided into sections:

Table 53. Powerdown sectioning

Section	Block
Print Engine Pipeline SubSystem (Section 0)	CDU
	CFU
	LBD
	SFU
	TE
	TFU
	HCU
	DNC
	DWU
	LLU
	PHI
CPU-DRAM (Section 1)	DRAM
	CPU/MMU
	DIU
	TIM
	ROM
	LSS Interface
Comms SubSystem (Section 2)	USB
	ISI
	DMA Ctrl
	GPIO
	PSS
	ICU

16.1.1 Sleep mode

Each section can be put into sleep mode by setting the corresponding bit in the *SleepModeEnable* register. To re-enable the section the sleep mode bit needs to be cleared and then the section should be reset by writing to the relevant bit in the *ResetSection* register. Each block within the section should then be re-configured by the CPU.

If the CPU system is put into sleep mode, the SoPEC device will remain in sleep mode until a system level reset is initiated from the reset pin, or a wakeup reset by the SCB block as a result of activity on either the

USB or ISI bus. If all sections are put into sleep mode, then only a system level reset initiated by the reset pin will re-activate the SoPEC device.

Like all software resets in SoPEC the *ResetSection* register is active-low i.e. a 0 should be written to each bit position requiring a reset. The *ResetSection* register is self-reseting.

16.2   RESET SOURCE

The SoPEC device can be reset by a number of sources. When a reset from an internal source is intiated the reset source register (*ResetSrc*) stores the reset source value. This register can then be used by the CPU to determine the type of boot sequence required.

16.3   CLOCK RELATIONSHIP

The crystal oscillator excites a 32MHz crystal through the xtalin and xtalout pins. The 32MHz output is used by the PLL to derive the master VCO frequency of 960MHz. The master clock is then divided to produce 320MHz clock (*clk320*), 160MHz clock (*clk160*), 106MHz clock (*clk106*) and 48MHz (*clk48*) clock sources.

The phase relationship of each clock from the PLL will be defined. The relationship of internal clocks *clk320, clk106, clk48* and *clk160* to *xtalin* will be undefined. The clock tree generation should create insertion delays so as to compensate for the phase difference of the clocks leaving the PLL. At the output of the clock block, the skew between each *pclk* domain (*pclk_section[3:0]* and *jclk*) should be within skew tolerances of their respective domains (defined as less than the hold time of a D-type flip flop).

The skew between *doclk* and *phiclk* should also be less than the skew tolerances of their respective domains.

The *usbclk* is derived from the PLL output and has no relationship with the other clocks in the system and is considered asynchronous.

There is no skew requirement between the *pclk* domains and the *doclk* and *phiclk* domains, they are considered essentially asynchronous to each other.
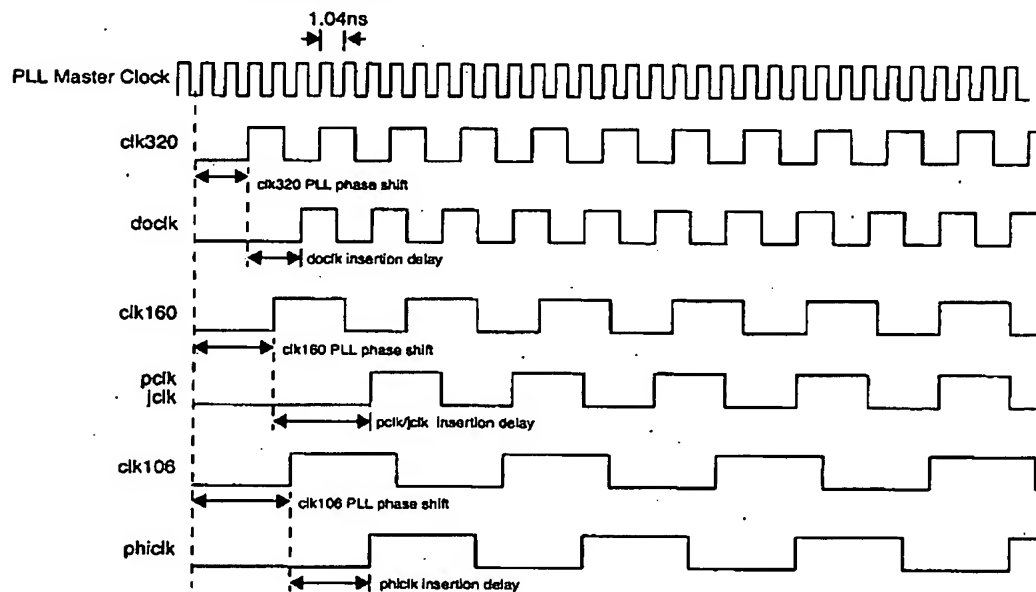

Figure 53. SoPEC clock relationship

16.4   IMPLEMENTATION

16.4.1 Definitions of I/O

Table 54. CPR I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
xtalin	1	In	Crystal input, direct from IO pin.
xtalout	1	Out	Crystal output, direct to IO pin.
pclk_section[2:0]	3	Out	System clocks for each section
phiclk	1	Out	Printhead interface clock (doclk/3) for the PHI block
doclk	1	Out	Data out clock (2x pclk) for the PHI block
jclk	1	Out	Gated version of system clock used to clock the JPEG decoder core in the CDU
usbclk	1	Out	USB clock at 3 times the crystal input frequency, nominally at 48 Mhz
jclk_enable	1	In	Gating signal for jclk.
reset_n	1	In	Reset signal from the *reset_n* pin
usb_cpr_reset_n	1	In	Reset signal from the USB block
isi_cpr_reset_n	1	In	Reset signal from the ISI block
tim_cpr_reset_n	1	In	Reset signal from watch dog timer.
prst_n_section[2:0]	3	Out	System resets for each section, synchronous active low
phirst_n	1	Out	Reset for PHI block, synchronous to *phiclk*
dorst_n	1	Out	Reset for PHI block, synchronous to *doclk*
jrst_n	1	Out	Reset for JPEG decoder core in CDU block, synchronous to *jclk*
usbrst_n	1	Out	Reset for the USB block, synchronous to *usbclk*
Test Input			
test_clk	1	In	Test clock direct from external pin, for use in production test (scan test)
test_enable	1	In	Test enable. Direct from external pin. When high production test mode is enabled.
CPU Interface			
cpu_adr[3:2]	2	In	CPU address bus. Only 2 bits are required to decode the address space for the CPR block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpr_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_cpr_sel	1	In	Block select from the CPU. When *cpu_cpr_sel* is high both *cpu_adr* and *cpu_dataout* are valid
cpr_cpu_rdy	1	Out	Ready signal to the CPU. When *cpr_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *cpr_cpu_data* is valid.
cpr_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpr_cpu_debug_valid	1	Out	Debug Data valid on *cpr_cpu_data* bus. Active high

Table 54. CPR I/O definition

Port name	Pins	I/O	Description
Miscellaneous			
pwr_sleep_mode[2:0]	3	Out	Sleep mode section select

16.4.2 Configuration registers

The configuration registers in the CPR are programmed via the CPU interface. Refer to section 11.4 on page 69 for a description of the protocol and timing diagrams for reading and writing registers in the CPR. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the CPR. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cpr_pcu_data*. Table 55 lists the configuration registers in the CPR block.

The CPR block will only allow supervisor data mode accesses (i.e. *cpu_acode[1:0]* = *SUPERVISOR_DATA*). All other accesses will result in *cpr_cpu_berr* being asserted .

Table 55. CPR Register Map

Address CPR_base+	Register	#bits	Reset	Description
0x00	SleepModeEnable	3	0x0	Sleep Mode enable, when high a section of logic has is powerdown. Each bit controls a section
0x04	ResetSrc	4	0x0[a]	Reset Source register, indicating the source of the last reset. Bit 0 - External Reset Bit 1 - USB wakeup reset Bit 2 - ISI wakeup reset Bit 3 - Watchdog timer reset
0x08	ResetSection	3	0x7	Active-low synchronous reset for each section, self-resetting.
0x0C	DebugSelect	6	0x00	Debug address select. Indicates the address of the register to report on the *cpr_cpu_data* bus when it is not otherwise being used.
PLL Control (Asynchronous reset registers)				
0x10	PLLTuneBits	10	0x23E	PLL tuning bits
0x14	PLLRangeA	4	0xF	PLLOUT A frequency selector (defaults to 600Mhz to1250Mhz)
0x18	PLLRangeB	3	0x7	PLLOUT B frequency selector (defaults to 600Mhz to1250Mhz)
0x1C	PLLMultiplier	5	0x25	PLL multiplier selector, defaults to *refclk* x 20

a. Reset value depends on reset source. External reset shown.
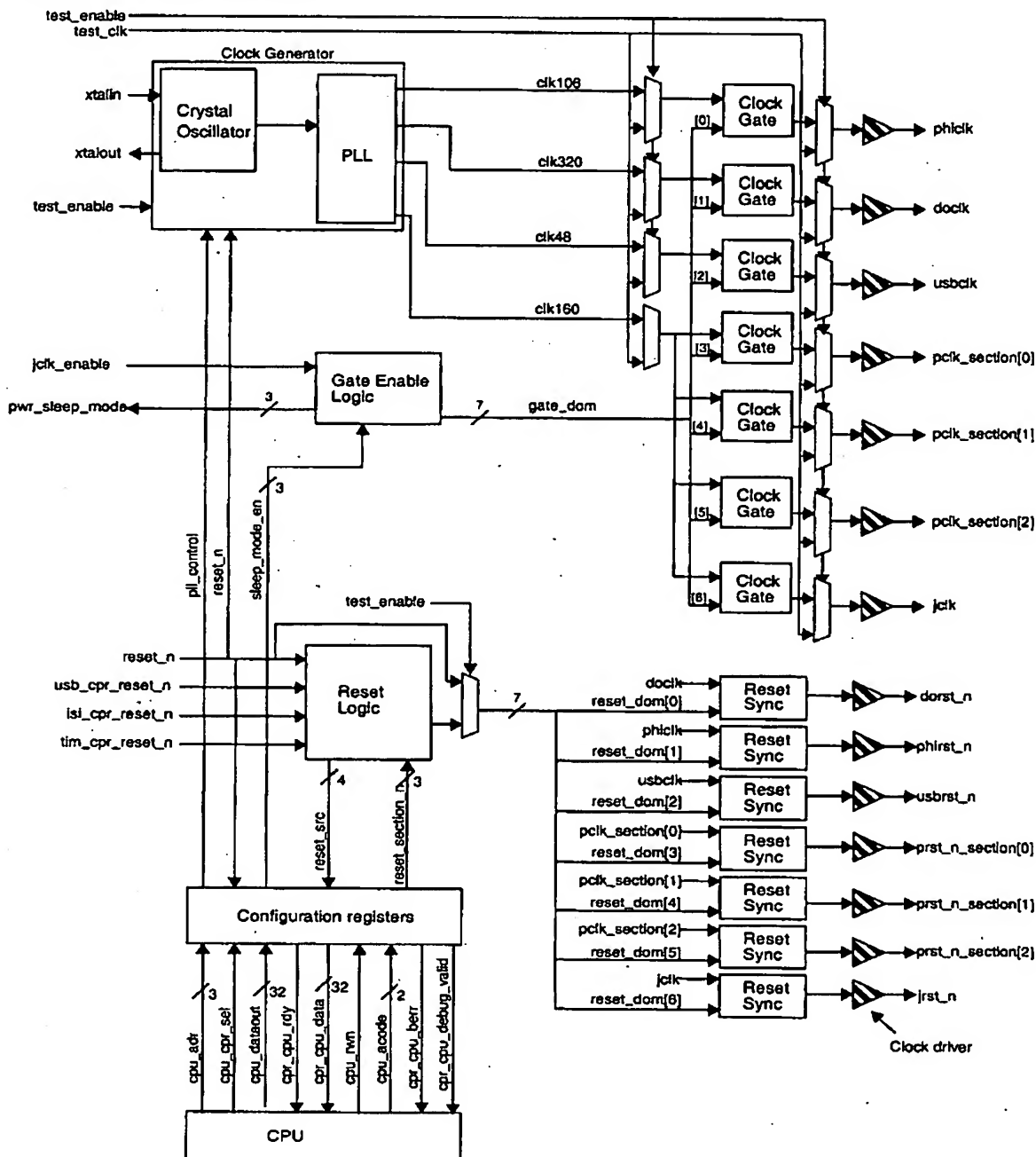
16.4.3   CPR Sub-block partition

Figure 54. CPR block partition

16.4.4 Sync reset

The reset synchronizer retimes an asynchronous reset signal to the clock domain that it resets. The circuit prevents the inactive edge of reset occurring when the clock is rising
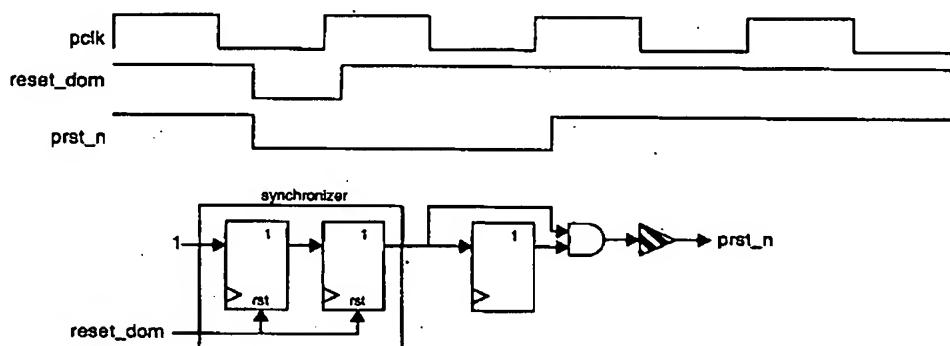

Figure 55. Reset synchronizer logic

16.4.5 Reset generator logic

The reset generator logic is used to determine which clock domains should be reset, based on configured reset values (*reset_section_n*), the external reset (*reset_n*), watchdog timer reset (*tim_cpr_reset_n*) and resets from the SCB block (*isi_cpr_reset_n, usb_cpr_reset_n*). The reset direct from the IO pin (*reset_n*) is synchronized and de-glitched before feeding the reset logic.

Resets from the SCB block reset everything except its own section (section 2), this allows data to be stored in the PSS block for use after a SCB powerup initiated reset.

Table 56. Reset domains

Reset signal	Domain
reset_dom[0]	doclk domain
reset_dom[1]	phiclk domain
reset_dom[2]	usbclk domain
reset_dom[3]	Section 0 pclk domain
reset_dom[4]	Section 1 pclk domain
reset_dom[5]	Section 2 pclk domain
reset_dom[6]	jclk domain

The logic is given by
```
if (reset_n == 0) then
    reset_dom[6:0] = 0x00        // reset everything
    reset_src[3:0] = 0x01
elsif (usb_cpr_reset_n == 0) then
    reset_dom[6:0] = 0x20        // all except comms domain
    reset_src[3:0] = 0x02
elsif (isi_cpr_reset_n == 0) then
    reset_dom[6:0] = 0x20        // all except comms domain
    reset_src[3:0] = 0x04
elsif (tim_cpr_reset_n == 0) then
    reset_dom[6:0] = 0x00        // reset everything
    reset_src[3:0] = 0x08
```

```
else
    // propagate resets from reset section register
    reset_dom[5:0] = 0x3F
    if (reset_section_n[0] == 0) then
        reset_dom[3] = 0
    if (reset_section_n[1] == 0) then
        reset_dom[4] = 0
    if (reset_section_n[2] == 0) then
        reset_dom[5] = 0
```

16.4.6   Gate enable logic

The gate enable logic is a combinational logic block used to generate gating signals for each of SoPECs clock domains. The gate enable (*gate_domain*) is generated based on the configured *sleep_mode_en* and the internally generated *jclk_enable* signal.

The logic is given by

```
// clock gating for sleep modes
gate_dom[5:3] = 0x7 // default to on
for (i=0 ;i < 3 ; i++){
    if (sleep_mode_en[i] == 1) then
        gate_dom[i+3] = 0
        pwr_sleep_mode[i] == 1
}
// jclk and remaining
gate_dom[2:0] = 0x7
gate_dom[6]   = ~(jclk_enable)
```

16.4.7   Clock gate logic

The clock gate logic is used to safely gate clocks without generating any glitches on the gated clock. When the enable is high the clock is active otherwise the clock is gated.
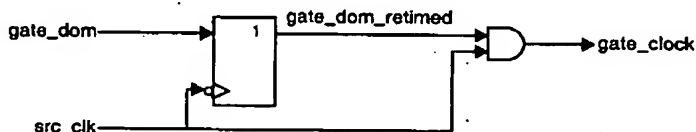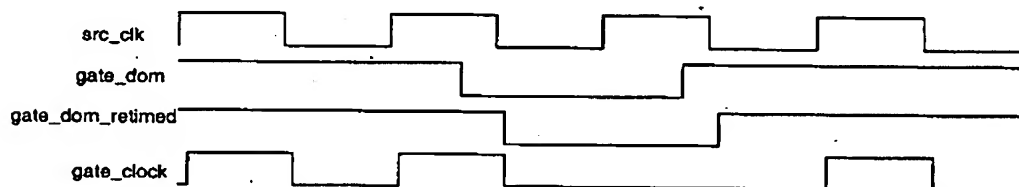

Figure 56. Clock gate logic diagram

16.4.8   Clock generator Logic

The clock generator block contains the PLL, crystal oscillator, clock dividers and associated control and test logic. The PLL VCO frequency is at 960Mhz locked to a 32 Mhz refclk generated by the crystal oscillator. In test mode the *xtalin* signal can be driven directly by the test clock generator, the test clock will be reflected on the *refclk* signal to the PLL.
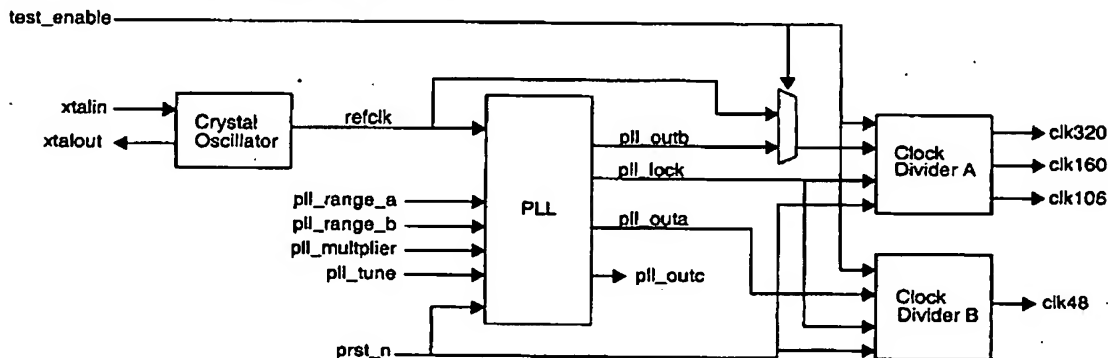

Figure 57. PLL and Clock divider logic

16.4.8.1   Clock divider A

The clock divider A block generate the 320Mhz, 160Mhz and 106Mhz clocks from the input 320Mhz clock (*pll_outb*) generated by the PLL. The divider flips flops are asynchronously reset by the *prst_n* signal. The divders are enabled only when the PLL has acquired lock as indicated by the *pll_lock* signal.

16.4.8.2   Clock divider B

The clock divider B block generate the 48Mhz clock from the input 96Mhz clock (*pll_outa*) generated by the PLL. The divider flips flops are asynchrously reset by the *prst_n* signal. The divders are enabled·only when the PLL has acquired lock as indicated by the *pll_lock* signal.

17 ROM Block

17.1 OVERVIEW

The ROM block interfaces to the CPU bus and contains the SoPEC boot code. The ROM block consists of the CPU bus interface, the ROM macro and the ChipID macro. The current ROM size is 16 KBytes implemented as a 4096 x32 macro. Access to the ROM is not cached because the CPU enjoys fast (no more than one cycle slower than a cache access), unarbitrated access to the ROM.

Each SoPEC device is required to have a unique ChipID which is set by blowing fuses at manufacture. IBM's 300mm ECID macro is to be used to implement the ChipID and this offers 112-bits of laser fuses. The exact number of fuse bits to be used for the ChipID will be determined later but all bits are made available to the CPU. The ECID macro allows all 112 bits to be read out in parallel and the ROM block will make all 112 bits available in the *FuseChipID[N]* registers which are readable by the CPU in supervisor mode only.

17.2 BOOT OPERATION

The are two boot scenarios for the SoPEC device namely after power-on and after being awoken from sleep mode. When the device is in sleep mode it is hoped that power will actually be removed from the DRAM, CPU and most other peripherals and so the program code will need to be freshly downloaded each time the device wakes up from sleep mode. In order to reduce the wakeup boot time (and hence the perceived print latency) certain data items are stored in the PSS block (see section 18). These data items include the SHA-1 hash digest expected for the program(s) to be downloaded, the master/slave SoPEC id and some configuration parameters (currently TBD). All of these data items are stored in the PSS by the CPU prior to entering sleep mode. The SHA-1 value stored in the PSS is calculated by the CPU by decrypting the signature of the downloaded program using the appropriate public key stored in ROM. This compute intensive decryption only needs to take place once as part of the power-on boot sequence - subsequent wakeup boot sequences will simply use the resulting SHA-1 digest stored in the PSS. Note that the digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

The CPU is expected to be in supervisor mode for the entire boot sequence described by the pseudocode below. Note that the boot sequence has not been finalised but is expected to be close to the following:

```
if (ResetSrc == 1)  then   // Reset was a power-on reset
   configure_sopec  // need to configure peris (USB, ISI, DMA, ICU etc.)
// Otherwise reset was a wakeup reset so peris etc. were already configured
PAUSE: wait until IrqSemaphore != 0   // i.e. wait until an interrupt has been serviced
if (IrqSemaphore == DMAChan0Msg) then
   parse_msg(DMAChan0MsgPtr)  // this routine will parse the message and take any
                             // necessary action e.g. programming the DMAChannel1 registers
elsif (IrqSemaphore == DMAChan1Msg) then  // program has been downloaded
   CalculatedHash = gen_sha1(ProgramLocn, ProgramSize)
   if (ResetSrc == 1)  then
      ExpectedHash = sig_decrypt(ProgramSig)
   else
      ExpectedHash = PSSHash
   if (ExpectedHash == CalculatedHash) then
      jmp(PrgramLocn)  // transfer control to the downloaded program
   else
      send_host_msg("Program Authentication Failed")
      goto PAUSE:
elsif (IrqSemaphore == timeout) then  // nothing has happened
   if (ResetSrc == 1)  then
```

```
                    sleep_mode()   // put SoPEC into sleep mode to be woken up by USB/ISI activity
             else   // we were woken up but nothing happened
                    reset_sopec(PowerOnReset)
       else
             goto PAUSE
```

The boot code places no restrictions on the activity of any programs downloaded and authenticated by it other than those imposed by the configuration of the MMU i.e. the principal function of the boot code is to authenticate that any programs downloaded by it are from a trusted source. It is the responsibility of the downloaded program to ensure that any code it downloads is also authenticated and that the system remains secure. The downloaded program code is also responsible for setting the SoPEC ISIId (see section 12.7 for a description of the ISIId) in a multi -SoPEC system. See the "SoPEC Security Overview" document [9] for more details of the SoPEC security features.

17.3   IMPLEMENTATION

17.3.1   Definitions of I/O

Table 57. ROM Block I/O

Port name	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
pclk	1	In	Global clock
CPU Interface			
cpu_adr[15:2]	14	In	CPU address bus. Only 14 bits are required to decode the address space for this block.
rom_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_rom_sel	1	In	Block select from the CPU. When *cpu_rom_sel* is high *cpu_adr* is valid
rom_cpu_rdy	1	Out	Ready signal to the CPU. When *rom_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on *rom_cpu_data* is valid.
rom_cpu_berr	1	Out	ROM bus error signal to the CPU indicating an invalid access.

17.3.2   Configuration registers

The ROM block will only allow read accesses to the *FuseChipID* registers with supervisor data space permissions (i.e. *cpu_acode[1:0]* = 11). All other accesses of the *FuseChipID* registers will result in *rom_cpu_berr* being asserted. The ROM block allows all read accesses to the ROM itself (i.e supervisor or

user, data or program accesses). The CPU subsystem bus slave interface is described in more detail in section 9.4.3.

Table 58. ROM Block Register Map

Address ROM Base +	Register	bits	Reset	Description
0x8000 to 0x8004	FuseChipID[N]	32	n/a	Value of corresponding fuse bits. (Read only)

17.3.3 Sub-Block Partition

IBM offer two variants of their ROM macros; A high performance version (ROMHD) and a low power version (ROMLD). It is likely that the low power version will be used unless some implementation issue requires the high performance version. Both versions offer the same bit density. The sub-block partition diagram below does not include the clocking and test signals for the ROM or ECID macros. The CPU subsystem bus interface is described in more detail in section 11.4.3.
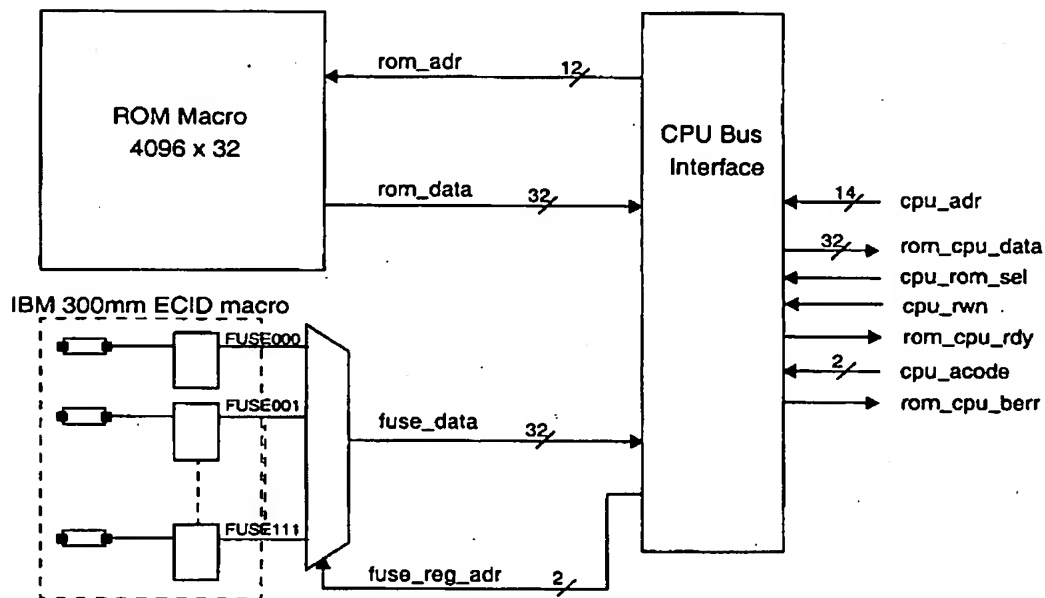

Figure 58. Sub-block partition of the ROM block

17.3.4 Sub-block signal definition

Table 59. ROM Block Internal signals

Port name	Width	Description
Clocks and Resets		
prst_n	1	Global reset. Synchronous to pclk, active low.

Table 59. ROM Block Internal signals

Port name	Width	Description
pclk	1	Global clock
Internal Signals		
rom_adr[11:0]	12	ROM address bus
rom_sel	1	Select signal to the ROM macro instructing it to access the location at *rom_adr*
rom_oe	1	Output enable signal to the ROM block
rom_data[31:0]	32	Data bus from the ROM macro to the CPU bus interface
rom_dvalid	1	Signal from the ROM macro indicating that the data on *rom_data* is valid for the address on *rom_adr*
fuse_data[31:0]	32	Data from the *FuseChipID[N]* register addressed by *fuse_reg_adr*
fuse_reg_adr[1:0]	2	Indicates which of the *FuseChipID* registers is being addressed

18 Power Safe Storage (PSS) Block

18.1 OVERVIEW

The PSS block provides 128 bytes of storage space that will maintain its state when the rest of the SoPEC device is in sleep mode. The PSS is expected to be used primarily for the storage of decrypted signatures associated with downloaded programmed code but it can also be used to store any information that needs to survive sleep mode (e.g. configuration details). Note that the signature digest only needs to be stored in the PSS before entering sleep mode and the PSS can be used for temporary storage of any data at all other times.

Prior to entering sleep mode the CPU should store all of the information it will need on exiting sleep mode in the PSS. On emerging from sleep mode the boot code in ROM will read the *ResetSrc* register in the CPR block to determine which reset source caused the wakeup. The reset source information indicates whether or not the PSS contains valid stored data, and the PSS data determines the type of boot sequence to execute. If for any reason a full power-on boot sequence should be performed (e.g. the printer driver has been updated) then this is simply achieved by initiating a full software reset.

18.2 IMPLEMENTATION

The storage area of the PSS block will be implemented as a 128-byte register array. The array is located from PSS_base through to PSS_base+0x7F in the address map. The PSS block will only allow read or write accesses with supervisor data space permissions (i.e. *cpu_acode[1:0]* = 11). All other accesses will result in *pss_cpu_berr* being asserted. The CPU subsystem bus slave interface is described in more detail in section 11.4.3.

18.2.1 Definitions of I/O

Table 60. PSS Block I/O

Portname	Pins	I/O	Description
Clocks and Resets			
prst_n	1	In	Global reset. Synchronous to pclk, active low.
pclk	1	In	Global clock
CPU Interface			
cpu_adr[6:2]	5	In	CPU address bus. Only 5 bits are required to decode the address space for this block.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
pss_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_pss_sel	1	In	Block select from the CPU. When *cpu_pss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
pss_cpu_rdy	1	Out	Ready signal to the CPU. When *pss_cpu_rdy* is high it indicates the last cycle of the access. For a read cycle this means the data on *pss_cpu_data* is valid.
pss_cpu_berr	1	Out	PSS bus error signal to the CPU indicating an invalid access.

19 Low Speed Serial Interface (LSS)

19.1 OVERVIEW

The Low Speed Serial Interface (LSS) provides a mechanism for the internal SoPEC CPU to communicate with external QA chips via two independent LSS buses. The LSS communicates through the GPIO block to the QA chips. This allows the QA chip pins to be reused in multi-SoPEC environments. The LSS Master system-level interface is illustrated in Figure 59. Note that multiple QA chips are allowed on each LSS bus.
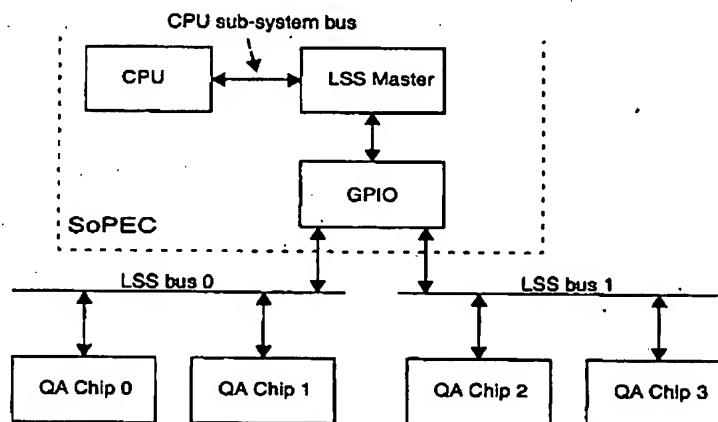

Figure 59. LSS master system-level interface

19.2 QA COMMUNICATION

The SoPEC data interface to the QA Chips is a low speed, 2 pin, synchronous serial bus. Data is transferred to the QA chips via the *lss_data* pin synchronously with the *lss_clk* pin. When the *lss_clk* is high the data on *lss_data* is deemed to be valid. Only the LSS master in SoPEC can drive the *lss_clk* pin, this pin is an input only to the QA chips. The LSS block must be able to interface with an open-collector pull-up bus. This means that when the LSS block should transmit a logical zero it will drive 0 on the bus, but when it should transmit a logical 1 it will leave high-impedance on the bus (i.e. it doesn't drive the bus). If all the agents on the LSS bus adhere to this protocol then there will be no issues with bus contention.

The LSS block controls all communication to and from the QA chips. The LSS block is the bus master in all cases. The LSS block interprets a command register set by the SoPEC CPU, initiates transactions to the QA chip in question and optionally accepts return data. Any return information is presented through the configuration registers to the SoPEC CPU. The LSS block indicates to the CPU the completion of a command or the occurrence of an error via an interrupt.

19.2.1 Start and stop conditions

All transmissions on the LSS bus are initiated by the LSS master issuing a START condition and terminated by the LSS master issuing a STOP condition. START and STOP conditions are always generated by the LSS master. As illustrated in Figure 60, a START condition corresponds to a high to low transition on

lss_data while *lss_clk* is high. A STOP condition corresponds to a low to high transition on *lss_data* while *lss_clk* is high.
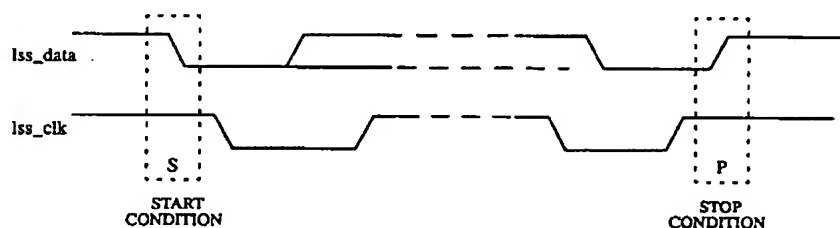

Figure 60. START and STOP conditions

19.2.2 Data transfer

Data is transferred on the LSS bus via a byte orientated protocol. Bytes are transmitted serially. Each byte is sent most significant bit (MSB) first through to least significant bit (LSB) last. One clock pulse is generated for each data bit transferred. Each byte must be followed by an acknowledge bit.

The data on the *lss_data* must be stable during the HIGH period of the *lss_clk* clock. Data may only change when *lss_clk* is low. A transmitter outputs data after the falling edge of *lss_clk* and a receiver inputs the data at the rising edge of *lss_clk*. This data is only considered as a valid data bit at the next *lss_clk* falling edge provided a START or STOP is not detected in the period before the next *lss_clk* falling edge. All clock pulses are generated by the LSS block. The transmitter releases the *lss_data* line (high) during the acknowledge clock pulse (ninth clock pulse). The receiver must pull down the *lss_data* line during the acknowledge clock pulse so that it remains stable low during the HIGH period of this clock pulse.

Data transfers follow the format shown in Figure 61. The first byte sent by the LSS master after a START condition is a primary id byte, where bits 7-2 form a 6-bit primary id (0 is a global id and will address all QA Chips on a particular LSS bus), bit 1 is an even parity bit for the primary id, and bit 0 forms the read/write sense. Bit 0 is high if the following command is a read to the primary id given or low for a write command to that id. An acknowledge is generated by the QA chip(s) corresponding to the given id (if such a chip exists) by driving the *lss_data* line low synchronous with the LSS master generated ninth *lss_clk*.
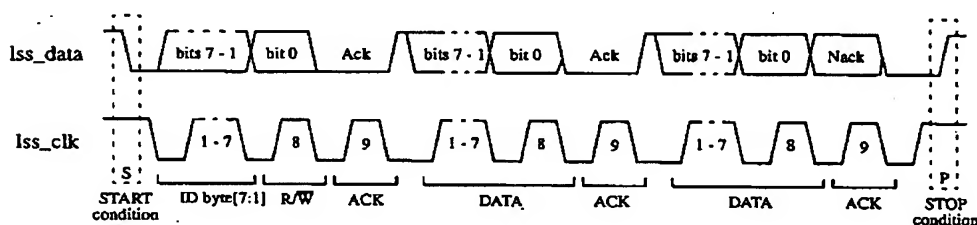

Figure 61. LSS transfer of 2 data bytes

19.2.3 Write procedure

The protocol for a write access to a QA Chip over the LSS bus is illustrated in Figure 63 below. The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then trans-

mits the primary id byte with a 0 in bit 0 to indicate that the following command is a write to the primary id. An acknowledge is generated by the QA chip corresponding to the given primary id. The LSS master will clock out M data bytes with the slave QA Chip acknowledging each successful byte written. Once the slave QA chip has acknowledged the Mth data byte the LSS master issues a STOP condition to complete the transfer. The QA chip gathers the M data bytes together and interprets them as a command. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8]. Note that the QA chip is free to not acknowledge any byte transmitted. The LSS master should respond by issuing an interrupt to the CPU to indicate this error. The CPU should then generate a STOP condition on the LSS bus to gracefully complete the transaction on the LSS bus.
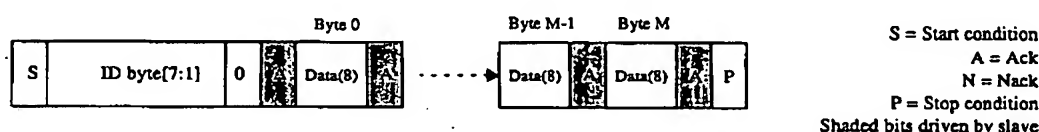

Figure 62. Example of LSS write to a QA Chip

19.2.4 Read procedure

The LSS master in SoPEC initiates the transaction by generating a START condition on the LSS bus. It then transmits the primary id byte with a 1 in bit 0 to indicate that the following command is a read to the primary id. An acknowledge is generated by the QA chip corresponding to the given primary id. The LSS master releases the *lss_data* bus and proceeds to clock the expected number of bytes from the QA chip with the LSS master acknowledging each successful byte read. The last expected byte is not acknowledged by the LSS master. It then completes the transaction by generating a STOP condition on the LSS bus. See QA Chip Interface Specification for more details on the format of the commands used to communicate with the QA chip[8].
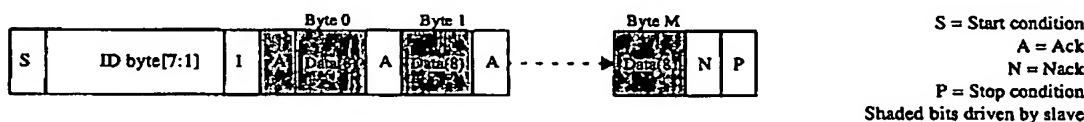
Figure 63. Example of LSS read from QA Chip

19.3   IMPLEMENTATION

A block diagram of the LSS master is given in Figure 64. It consists of a block of configuration registers that are programmed by the CPU and two identical LSS master units that generate the signalling protocols on the two LSS buses as well as interrupts to the CPU. The CPU initiates and terminates transactions on the LSS buses by writing an appropriate command to the command register, writes bytes to be transmitted to a fifo and reads bytes received from a fifo, and checks the sources of interrupts by reading status registers.

Figure 64. LSS block diagram

19.3.1   Definitions of IO

Table 61. LSS IO pins definitions

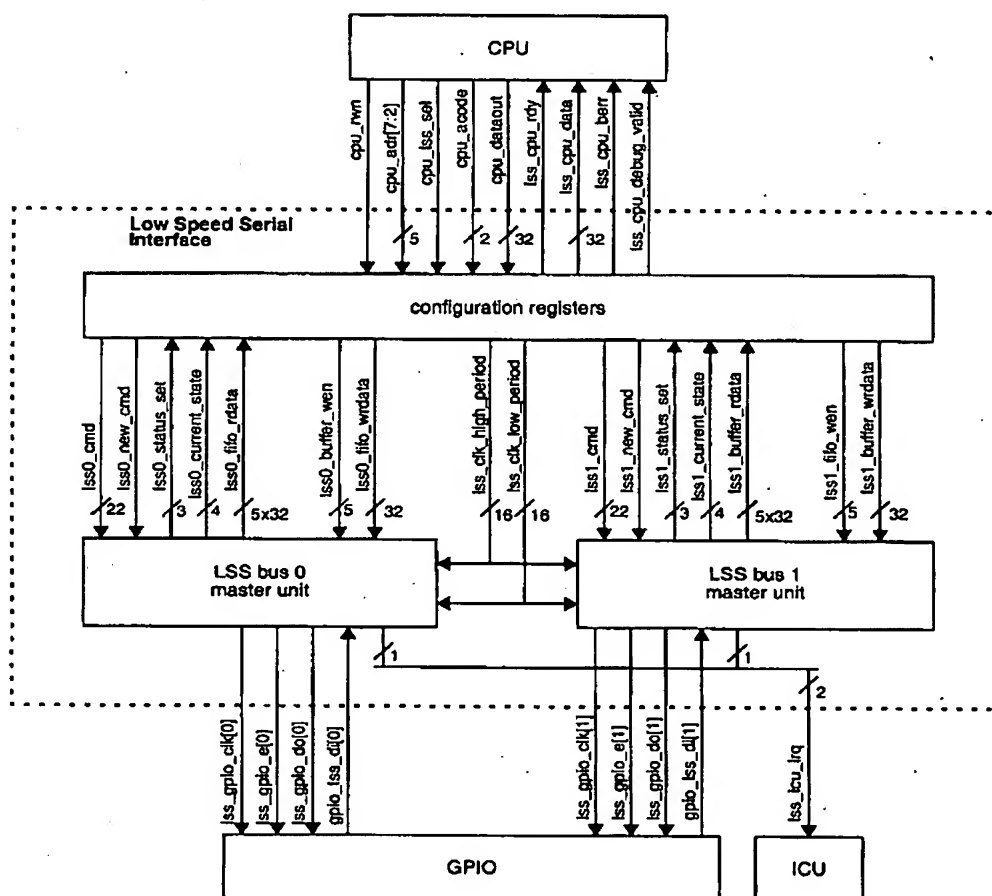| Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface			
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_adr[7:2]	5	In	CPU address bus. Only 6 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
cpu_acode[1:0]	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access
cpu_lss_sel	1	In	Block select from the CPU. When *cpu_lss_sel* is high both *cpu_adr* and *cpu_dataout* are valid
lss_cpu_rdy	1	Out	Ready signal to the CPU. When *lss_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the LSS block and for a read cycle this means the data on *lss_cpu_data* is valid.
lss_cpu_berr	1	Out	LSS bus error signal to the CPU.
lss_cpu_data[31:0]	32	Out	Read data bus to the CPU
lss_cpu_debug_valid	1	Out	Active high. Indicates the presence of valid debug data on *lss_cpu_data*.
GPIO for LSS buses			
lss_gpio_do[1:0]	2	Out	LSS bus data output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
gpio_lss_di[1:0]	2	In	LSS bus data input Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_e[1:0]	2	Out	LSS bus data output enable, active high Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
lss_gpio_clk[1:0]	2	Out	LSS bus clock output Bit 0 - LSS bus 0 Bit 1 - LSS bus 1
ICU Interface			
lss_icu_irq[1:0]	2	Out	LSS interrupt requests Bit 0 - interrupt associated with LSS bus 0 Bit 1 - interrupt associated with LSS bus 1

19.3.2 Configuration registers

The configuration registers in the LSS block are programmed via the CPU interface. Refer to section 11.4 on page 69 for the description of the protocol and timing diagrams for reading and writing registers in the LSS block. Note that since addresses in SoPEC are byte aligned and the CPU only supports 32-bit register reads and writes, the lower 2 bits of the CPU address bus are not required to decode the address space for the LSS block. Table 62 lists the configuration registers in the LSS block. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *lss_cpu_data*.

The input *cpu_acode* signal indicates whether the current CPU access is supervisor, user, program or data. The configuration registers in the LSS block can only be read or written by a supervisor data access, i.e. when *cpu_acode* equals b11. If the current access is a supervisor data access then the LSS responds by asserting *lss_cpu_rdy* for a single clock cycle.

If the current access is anything other than a supervisor data access, then the LSS generates a bus error by asserting *lss_cpu_berr* for a single clock cycle instead of *lss_cpu_rdy* as shown in section 11.4 on page 69. A write access will be ignored, and a read access will return zero.

Table 62. LSS Control Registers

Address (LSS base +)	Register	#bits	Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the LSS.
0x04	LssClockHighPeriod	16	0x00C8	High period of *lss_clk* expressed as a number of *pclk* cycles. Transmission over the LSS bus is at a nominal rate of 400kHz, corresponding to a high period of 200 *pclk* (160Mhz) cycles for a 50/50 duty cycle.
0x08	LssClockLowPeriod	16	0x00C8	Low period of *lss_clk* expressed as a number of *pclk* cycles. Transmission over the LSS bus is at a nominal rate of 400kHz, corresponding to a low period of 200 *pclk* (160Mhz) cycles for a 50/50 duty cycle.
LSS bus 0 registers				
0x10	Lss0IntStatus	3	0x0	LSS bus 0 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 0 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 0 A 1 in a bit of *lss0_status_set* signal causes the corresponding bit in *Lss0IntStatus* register to be set. All the bits in *Lss0IntStatus* are cleared when the *Lss0Cmd* register gets written to. **(Read only register)**
0x14	Lss0CurrentState	4	0x0	Gives the current state of the LSS bus 0 state machine. **(Read only register).** (Encoding will be specified upon state machine implementation)
0x18	Lss0Cmd	22	0x00_0000	Command register defining sequence of events to perform on LSS bus 0 before interrupting CPU. A write to this register causes all the bits in the *Lss0IntStatus* register to be cleared as well as generating a *lss0_new_cmd* pulse.

Table 62. LSS Control Registers

Address (LSS base)	Register	#bits	Reset	Description
0x1C - 0x2C	Lss0fifo[4:0]	5x32	0x0000_0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
LSS bus 1 registers				
0x30	Lss1IntStatus	3	0x0	LSS bus 1 interrupt status registers Bit 0 - command completed successfully Bit 1 - error during processing of command, not -acknowledge received after transmission of primary id byte on LSS bus 1 Bit 2 - error during processing of command, not -acknowledge received after transmission of data byte on LSS bus 1 A 1 in a bit of *lss1_status_set* signal causes the corresponding bit in *Lss1IntStatus* register to be set. All the bits in *Lss1IntStatus* are cleared when the *Lss1Cmd* register gets written to. (Read only register)
0x34	Lss1CurrentState	4	0x0	Gives the current state of the LSS bus 1 state machine. (Read only register) (Encoding will be specified upon state machine implementation)
0x38	Lss1Cmd	22	0x00_0000	Command register defining sequence of events to perform on LSS bus 1 before interrupting CPU. A write to this register causes all the bits in the *Lss1IntStatus* register to be cleared as well as generating a *lss1_new_cmd* pulse.
0x3C - 0x4C	Lss1Buffer[4:0]	5x32	0x0000_0000	LSS Data buffer. Should be filled with transmit data before transmit command, or read data bytes received after a valid read command.
Debug registers				
0x50	LssDebugSel	5	0x00	Selects register for debug output. This value is used as the input to the register decode logic instead of *cpu_adr[6:2]* when the LSS block is not being accessed by the CPU, i.e. when *cpu_lss_sel* is 0. The output *lss_cpu_debug_valid* is asserted to indicate that the data on *lss_cpu_data* is valid debug data. This data can be mutiplexed onto chip pins during debug mode.

19.3.2.1 LSS command registers

The LSS command registers define a sequence of events to perform on the respective LSS bus before issuing an interrupt to the CPU. There is a separate command register and interrupt for each LSS bus. The format of the command is given in Table 63. The CPU writes to the command register to initiate a sequence of events on an LSS bus. Once the sequence of events has completed or an error has occurred, an interrupt is sent back to the CPU.

Some example commands are:
- a single START condition (*Start* = 1, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 0)
- a single STOP condition (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 0, *Stop* = 1)
- a START condition followed by transmission of the id byte (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 0, *Stop* = 0, *IdByte* contains primary id byte)

- a write transfer of 20 bytes from the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 0, *Stop* = 0, *TxRxByteCount* = 20)
- a read transfer of 8 bytes into the data buffer (*Start* = 0, *IdByteEnable* = 0, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack* = 0, *Stop* = 0, *TxRxByteCount* = 8)
- a complete read transaction of 16 bytes (*Start* = 1, *IdByteEnable* = 1, *RdWrEnable* = 1, *RdWrSense* = 1, *ReadNack* = 1, *Stop* = 1, *IdByte* contains primary id byte, *TxRxByteCount* = 16), etc.

The CPU can thus program the number of bytes to be transmitted or received (up to a maximum of 20) on the LSS bus before it gets interrupted. This allows it to insert arbitrary delays in a transfer at a byte boundary. For example the CPU may want to transmit 30 bytes to a QA chip but insert a delay between the 20th and 21st bytes sent. It does this by first writing 20 bytes to the data buffer. It then writes a command to generate a START condition, send the primary id byte and then transmit the 20 bytes from the data buffer. When interrupted by the LSS block to indicate successful completion of the command the CPU can then write the remaining 10 bytes to the data buffer. It can then wait for a defined period of time before writing a command to transmit the 10 bytes from the data buffer and generate a STOP condition to terminate the transaction over the LSS bus.

An interrupt to the CPU is generated for one cycle when any bit in *LssNIntStatus* is set. The CPU can read *LssNIntStatus* to discover the source of the interrupt and can clear a bit in *LssNIntStatus* by writing a 1 to the corresponding bit in *LssNIntStatus* register. Alternatively the CPU can start a new command which will automatically reset all *LssNIntStatus* bits.

Table 63. LSS command register description

bit(s)	name	description
0	Start	When 1, issue a START condition on the LSS bus.
1	IdByteEnable	ID byte transmit enable: 1 - transmit byte in *IdByte* field 0 - Ignore byte in *IdByte* field
2	RdWrEnable	Read/write transfer enable: 0 - Ignore settings of *RdWrSense*, *ReadNack* and *TxRxByteCount* 1 - if RdWrSense is 0, then perform a write transfer of *TxRxByteCount* bytes from the data buffer. if RdWrSense is 1, then perform a read transfer of *TxRxByteCount* bytes into the data buffer. Each byte should be acknowledged and the last byte received is acknowledged/not-acknowledged according to the setting of *ReadNack*.
3	RdWrSense	Read/write sense indicator: 0 - write 1 - read
4	ReadNack	Indicates, for a read transfer, whether to issue an acknowledge or a not-acknowledge after the last byte received (indicated by *TxRxByteCount*). 0 - issue acknowledge after last byte received 1 - issue not-acknowledge after last byte received.
5	Stop	When 1, issue a STOP condition on the LSS bus.
7:6	reserved	Must be 0
15:8	IdByte	Byte to be transmitted if *IdByteEnable* is 1. Bit 8 corresponds to the LSB.
20:16	TxRxByteCount	Number of bytes to be transmitted from the data buffer or the number of bytes to be received into the data buffer. The maximum value that should be programmed is 20, as the size of the data buffer is 20 bytes.

The data buffer is implemented in the LSS master block. When the CPU writes to the *LssNBuffer* registers the data written is presented to the LSS master block via the *lssN_buffer_wrdata* bus and configuration registers block pulses the *lssN_buffer_wen* bit corresponding to the register written. For example if *LssN-*

Buffer[2] is written to *lssN_buffer_wen[2]* will be pulsed. When the CPU reads the *LssNBuffer* registers the configuration registers block reflect the *lssN_buffer_rdata* bus back to the CPU.

19.3.3   LSS master unit

The LSS master unit is instantiated for both LSS bus 0 and LSS bus 1. It controls transactions on the LSS bus by means of the state machine shown in Figure 65, which interprets the commands that are written by the CPU. It also contains a single 20 byte data buffer used for transmitting and receiving data.

The CPU can write data to be transmitted on the LSS bus by writing to the *LssNBuffer* registers. It can also read data that the LSS master unit receives on the LSS bus by reading the same registers. The LSS master always transmits or receives bytes to or from the data buffer in the same order. For example a transmit command

For a transmit command, *LssNBuffer[0][7:0]* gets transmitted first, then *LssNBuffer[0][15:8]*, *LssNBuffer[0][23:16]*, *LssNBuffer[0][31:24]*, *LssNBuffer[1][7:0]* and so on until *TxRxByteCount* number of bytes are transmitted. A receive command fills data to the buffer in the same order. Each new command the buffer start point is reset.

All state machine outputs, flags and counters are cleared on reset. After a reset the state machine remains in the *Idle* state until *lss_cmd_valid* equals 1. If the *Start* bit of the command is 0 the state machine proceeds directly to the *CheckIdByteEnable* state. If the *Start* bit is 1 it proceeds to the *GenerateStart* state and issues a START condition on the LSS bus.

In the *CheckIdByteEnable* state, if the *IdByteEnable* bit of the command is 0 the state machine proceeds directly to the *CheckRdWrEnable* state. If the *IdByteEnable* bit is 1 the state machine enters the *SendId-Byte* state and the byte in the *IdByte* field of the command is transmitted on the LSS. The *WaitForIdAck* state is then entered. If the byte is acknowledged, the state machine proceeds to the *CheckRdWrEnable* state. If the byte is not-acknowledged, the state machine proceeds to the *GenerateInterrupt* state and issues an interrupt to indicate a not-acknowledge was received after transmission of the primary id byte.

In the *CheckRdWrEnable* state, if the *RdWrEnable* bit of the command is 0 the state machine proceeds directly to the *CheckStop* state. If the *RdWrEnable* bit is 1, *count* is loaded with the value of the *TxRxByteCount* field of the command and the state machine enters either the *ReceiveByte* state if the *RdWrSense* bit of the command is 1 or the *TransmitByte* state if the *RdWrSense* bit is 0.

For a write transaction, the state machine keeps transmitting bytes from the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. If all the bytes are successfully transmitted the state machine proceeds to the *CheckStop* state. If the slave QA chip not-acknowledges a transmitted byte, the state machine indicates this error by issuing an interrupt to the CPU and then entering the *GenerateInterrupt* state.

For a read transaction, the state machine keeps receiving bytes into the data buffer, decrementing *count* after each byte transmitted, until *count* is 1. After each byte received the LSS master must issue an acknowledge. After the last expected byte (i.e. when count is 1) the state machine checks the *ReadNack* bit of the command to see whether it must issue an acknowledge or not-acknowledge for that byte. The *CheckStop* state is then entered.

In the *CheckStop* state, if the *Stop* bit of the command is 0 the state machine proceeds directly to the *GenerateInterrupt* state. If the *Stop* bit is 1 it proceeds to the *GenerateStop* state and issues a STOP condition on the LSS bus before proceeding to the *GenerateInterrupt* state. In both cases an interrupt is issued to indicate successful completion of the command.

The state machine then enters the *Idle* state to await the next command.

The CPU may abort the current transfer at any time by performing a write to the *Reset* register of the LSS block.

19.3.3.1 START and STOP generation

START and STOP conditions, which signal the beginning and end of data transmission, occur when the LSS master generates a falling and rising edge respectively on the data while the clock is high.

In the *GenerateStart* state, *lss_gpio_clk* is held high with *lss_gpio_e* remaining deasserted (so the data line is pulled high externally) for *LssClockHighPeriod pclk* cycles. Then *lss_gpio_e* is asserted and *lss_gpio_do* is pulled low (to drive a 0 on the data line, creating a falling edge) with *lss_gpio_clk* remaining high for another *LssClockHighPeriod pclk* cycles.

In the *GenerateStop* state, both *lss_gpio_clk* and *lss_gpio_do* are pulled low followed by the assertion of *lss_gpio_e* to drive a 0 while the clock is low. After *LssClockLowPeriod pclk* cycles, *lss_gpio_clk* is set high. After a further *LssClockHighPeriod pclk* cycles, *lss_gpio_e* is deasserted to release the data bus and create a rising edge on the data bus during the high period of the clock.

19.3.3.2 Clock pulse generation

The LSS master holds lss_gpio_clk high while the LSS bus is inactive. A clock pulse is generated for each bit transmitted or received over the LSS bus. It is generated by first holding *lss_gpio_clk* low for *LssClockLowPeriod pclk* cycles, and then high for *LssClockHighPeriod pclk* cycles.

19.3.3.3 Data reception

The input data, *gpio_lss_di*, is first synchronised to the *pclk* domain by means of two flip-flops clocked by *pclk*. The LSS master generates a clock pulse for each bit received. The output *lss_gpio_e* is deasserted on the falling edge of *lss_gpio_clk* to release the data bus. The value on the synchronised *gpio_lss_di* is sampled on the rising edge of *lss_gpio_clk* (the data should be averaged over a further 3 stage register to avoid possible glitch detection). The data is only considered as a valid bit at the next falling edge of *lss_gpio_clk* provided a START or STOP is not generated in the meantime.

In the *ReceiveByte* state, the state machine generates 8 clock pulses. On each rising edge of *lss_gpio_clk* the synchronised *gpio_lss_di* is sampled. The first bit sampled is *LssNBuffer[0][7]*, the second *LssNBuffer[0][6]*, etc to *LssNBuffer[0][0]*. For each byte received the state machine either sends an NAK or an ACK depending on the command configuration and the number of bytes received.

In the *SendNack* state the state machine generates a single clock pulse. *lss_gpio_e* is deasserted and the LSS data line is pulled high externally to issue a not-acknowledge.

In the *SendAck* state the state machine generates a single clock pulse. *lss_gpio_e* is asserted and a 0 driven on *lss_gpio_do* after *lss_gpio_clk* falling edge to issue an acknowledge.

19.3.3.4 Data transmission

The LSS master generates a clock pulse for each bit transmitted. Data is output on the LSS bus on the falling edge of *lss_gpio_clk*.

When the LSS master drives a logical zero on the bus it will assert *lss_gpio_e* and drive a 0 on *lss_gpio_do* after *lss_gpio_clk* falling edge. *lss_gpio_e* will remain asserted and *lss_gpio_do* will remain low until the next *lss_clk* falling edge.

When the LSS master drives a logical one *lss_gpio_e* should be deasserted at *lss_gpio_clk* falling edge and remain deasserted at least until the next *lss_gpio_clk* falling edge. This is because the LSS bus will be externally pulled up to logical one via a pull-up resistor.

In the *SendId byte* state, the state machine generates 8 clock pulses to transmit the byte in the *IdByte* field of the current valid command. On each falling edge of *lss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *IdByte[7]* is driven on the data bus, on the second falling edge *IdByte[6]* is driven out, etc.

In the *TransmitByte* state, the state machine generates 8 clock pulses to transmit the byte at the output of the transmit FIFO. On each falling edge of *lss_gpio_clk* a bit is driven on the data bus as outlined above. On the first falling edge *LssNBuffer[0][7]* is driven on the data bus, on the second falling edge *LssNBuffer[0][6]* is driven out, etc on to *LssNBuffer[0][7]* bits.

In the *WaitForAck* state, the state machine generates a single clock pulse. On the rising edge of *lss_gpio_clk* the synchronized *gpio_lss_di* is sampled. A 1 indicates an acknowledge and *ack_detect* is pulsed, a 0 indicates a not-acknowledge and *nack_detect* is pulsed.

19.3.3.5 Data rate control

The CPU can control the data rate by setting the clock period of the LSS bus clock by programming appropriate values in *LssClockHighPeriod* and *LssClockLowPeriod*. The default setting for both registers is 200 (*pclk* cycles) which corresponds to transmission rate of 400kHz on the LSS bus.

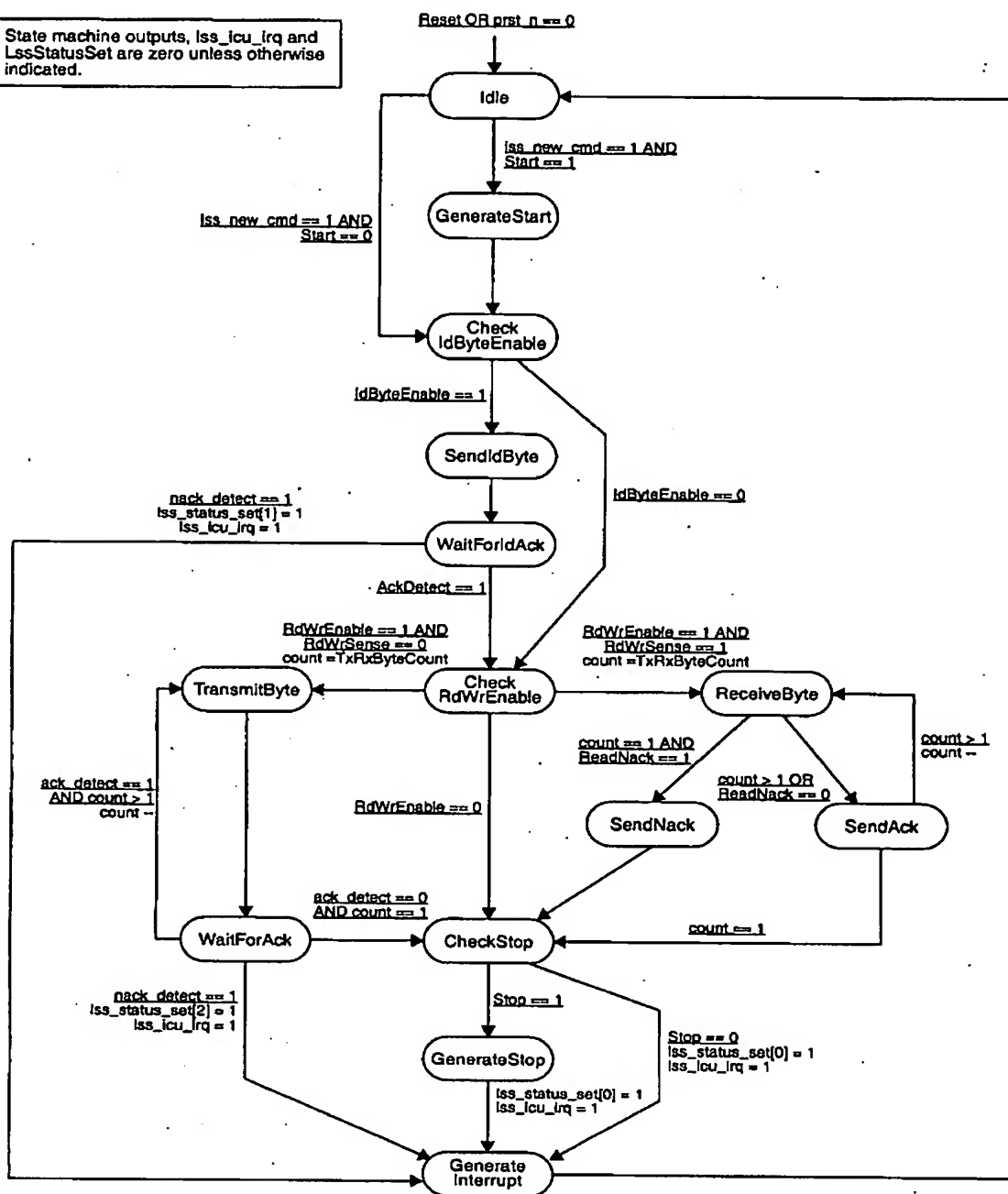State machine outputs, lss_icu_irq and LssStatusSet are zero unless otherwise indicated.

Reset OR prst_n == 0

Idle

lss_new_cmd == 1 AND
Start == 1

GenerateStart

lss_new_cmd == 1 AND
Start == 0

Check
IdByteEnable

IdByteEnable == 1

SendIdByte

IdByteEnable == 0

nack_detect == 1
lss_status_set[1] = 1
lss_icu_irq = 1

WaitForIdAck

AckDetect == 1

RdWrEnable == 1 AND
RdWrSense == 0
count = TxRxByteCount

RdWrEnable == 1 AND
RdWrSense == 1
count = TxRxByteCount

TransmitByte

Check
RdWrEnable

ReceiveByte

count == 1 AND
ReadNack == 1

count > 1
count --

ack_detect == 1
AND count > 1
count --

RdWrEnable == 0

count > 1 OR
ReadNack == 0

SendNack

SendAck

ack_detect == 0
AND count == 1

WaitForAck

CheckStop

count == 1

nack_detect == 1
lss_status_set[2] = 1
lss_icu_irq = 1

Stop == 1

GenerateStop

Stop == 0
lss_status_set[0] = 1
lss_icu_irq = 1

lss_status_set[0] = 1
lss_icu_irq = 1

Generate
Interrupt

Figure 65. LSS master state machine

DRAM SUBSYSTEM

20 DRAM Interface Unit (DIU)

20.1 OVERVIEW

Figure 66 shows how the DIU provides the interface between the on-chip 20 Mbit embedded DRAM and the rest of SoPEC. In addition to outlining the functionality of the DIU, this chapter provides a top-level overview of the memory storage and access patterns of SoPEC and the buffering required in the various SoPEC blocks to support those access requirements.

The main functionality of the DIU is to arbitrate between requests for access to the embedded DRAM and provide read or write accesses to the requesters. The DIU must also implement the initialisation sequence and refresh logic for the embedded DRAM.

The arbitration mechanism is a hierarchical timeslot mechanism providing guaranteed bandwidth and latency to each DIU requester, with unused slots re-allocated to provide best effort accesses. The arbitration scheme is fully programmable.

The interface between the DIU and the SoPEC requesters is similar to the interface on PEC1 i.e. separate control, read data and write data busses.

The embedded DRAM is used principally to store:
- CPU program code and data.
- PEP (re)programming commands.
- Compressed pages containing contone, bi-level and raw tag data and header information.
- Decompressed contone and bi-level data.
- Dotline store during a print.
- Print setup information such as tag format structures, dither matrices and dead nozzle information.
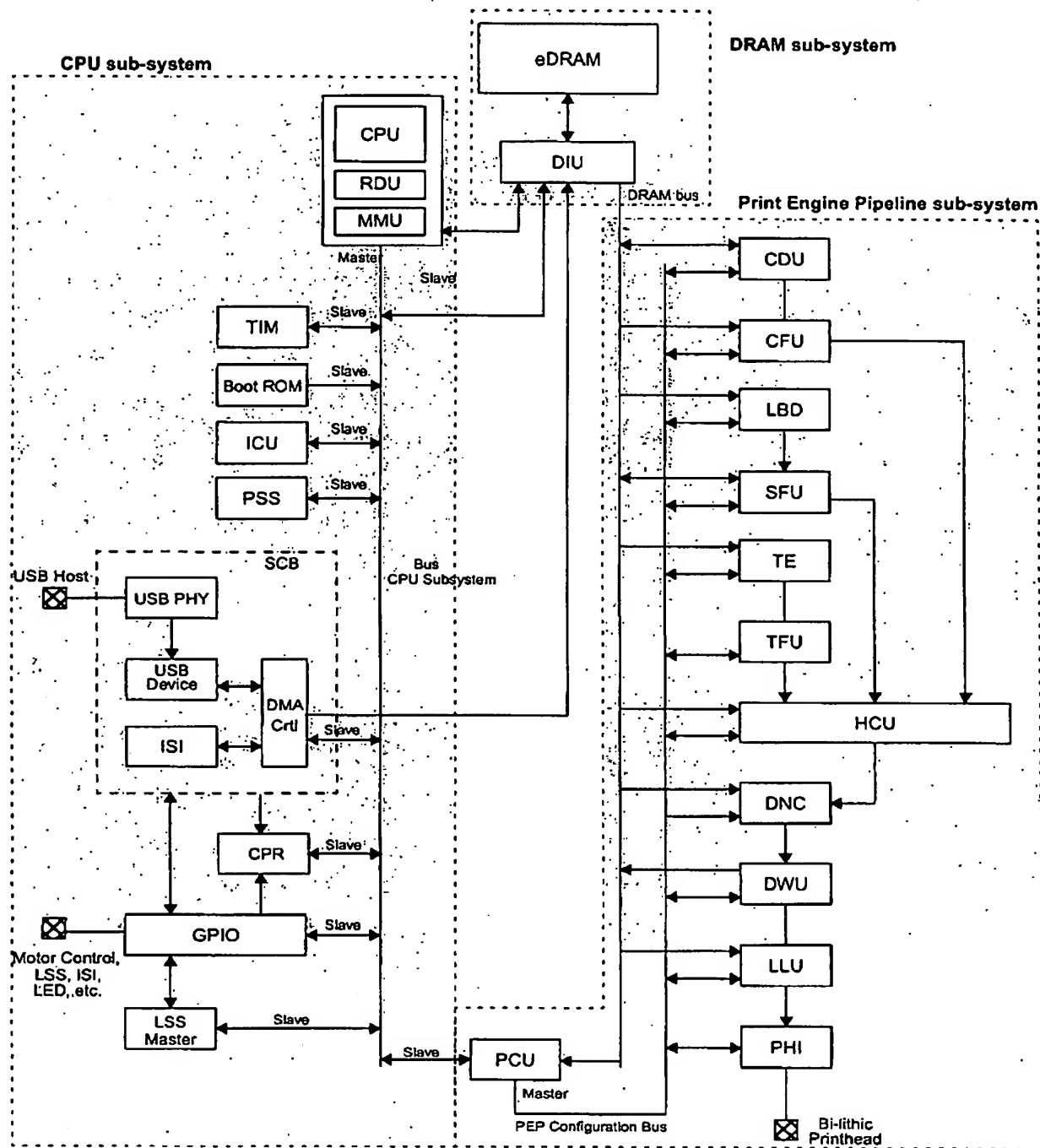
Figure 66. SoPEC System Top Level partition

20.2   IBM Cu-11 Embedded DRAM

20.2.1   Single bank

SoPEC will use the 1.5 V core voltage option in IBM's 0.13 μm class Cu-11 process.

The random read/write cycle time and the refresh cycle time is 3 cycles at 160 MHz [16]. An open page access will complete in 1 cycle if the page mode select signal is clocked at 320 MHz or 2 cycles if the page mode select signal is clocked every 160 MHz cycle. The page mode select signal will be clocked at 320 MHz in SoPEC. The DRAM word size is 256 bits.

Most SoPEC requesters will make single 256 bit DRAM accesses (see Section 20.4). These accesses will take 3 cycles as they are random accesses i.e. they will most likely be to a different memory row than the previous access.

The entire 20 Mbit DRAM will be implemented as a single memory bank. In Cu-11, the maximum single instance size is 16 Mbit. The first 1 Mbit tile of each instance contains an area overhead so the cheapest solution in terms of area is to have only 2 instances. 16 Mbit and 4Mbit instances would together consume an area of 14.63 mm^2 as would 2 times 10 Mbit instances. 4 times 5 Mbit instances would require 17.2 mm^2.

The instance size will determine the frequency of refresh. Each refresh requires 3 clock cycles. In Cu-11 each row consists of 8 columns of 256-bit words. This means that 16 Mbit requires 8192 rows. A complete DRAM refresh is required every 3.2 ms. This would mean a row would have to be refreshed every 62 cycles. Two times 10 Mbit instances would require a refresh every 100 clock cycles, if the instances are refreshed in parallel. Having 4 times 5 Mbit instances means a refresh is required only every 200 cycles.

The SoPEC DRAM will be constructed as two 10 Mbit instances implemented as a single memory bank.

20.3 SoPEC MEMORY USAGE REQUIREMENTS

The memory usage requirements for the embedded DRAM are shown in Table 64.

Table 64. Memory Usage Requirements

Block	Size	Description
Compressed page store	2048 Kbytes	Compressed data page store for Bi-level and contone data
Decompressed Contone Store	108 Kbyte	13824 lines with scale factor 6 = 2304 pixels, store 12 lines, 4 colors = 108 kB 13824 lines with scale factor 5 = 2765 pixels, store 12 lines, 4 colors = 130 kB
Spot line store	5.1 Kbyte	13824 dots/line so 3 lines is 5.1 kB
Tag Format Structure	55 Kbyte (384 dot line tags @ 1600 dpi) 12 Kbyte (2.5 mm tags @ 800 dpi)	55 kB in for 384 dot line tags 2.5 mm tags (1/10th inch) @ 1600 dpi require 160 dot lines = 160/384 x55 or 23 kB 2.5 mm tags @ 800 dpi require 80/384 x55 = 12 kB
Dither Matrix store	4 Kbytes	64x64 dither matrix is 4 kB 128x128 dither matrix is 16 kB 256x256 dither matrix is 64 kB
DNC Dead Nozzle Table	1.4 Kbytes	Delta encoded, (10 bit delta position + 6 dead nozzle mask) x% Dnozzle 5% dead nozzles requires (10+6)x 692 Dnozzles = 1.4 Kbytes
Dot-line store	319 Kbytes	Assume each color row is separated by 5 dot lines on the print head The dot line store will be 0+5+10...50+55 = 330 half dot lines + 48 extra half dot lines (4 per dot row) = 378 half dot lines = 319Kbytes
PCU Program code	8 Kbytes	1024 commands of 64 bits = 8 kB
CPU	64 Kbytes	Program code and data
TOTAL	2570 Kbytes (12 Kbyte TFS storage) 2613 Kbytes (55 Kbyte TFS)	

Note:

• Total storage of 2570 Kbytes will be reduced to 2560 Kbytes to align to 20 Mbit DRAM.

20.4   SoPEC MEMORY ACCESS PATTERNS

Table 65 shows a summary of the blocks on SoPEC requiring access to the embedded DRAM and their individual memory access patterns. Most blocks will access the DRAM in single 256-bit accesses. All accesses must be padded to 256-bits except for 64-bit CDU write accesses and CPU write accesses. Bits which should not be written are masked using the individual DRAM bit write inputs or byte write inputs, depending on the foundry. Using single 256-bit accesses means that the buffering required in the SoPEC DRAM requesters will be minimized.

Table 65. Memory access patterns of SoPEC DRAM Requesters

DRAM requester	Direction	Memory access pattern
CPU	R	Single 256-bit reads.
	W	Single 32-bit, 16-bit or 8-bit writes.
SCB	W	Single 256-bit writes.
CDU	R	Single 256-bit reads of the compressed contone data.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining bits write masked. The access time for this 4 word page mode burst is 3 + 1 + 1 +1 = 6 cycles if the page mode select signal is clocked at 320 MHz.
CFU	R	Single 256 bit reads.
LBD	R	Single 256 bit reads.
SFU	R	Separate single 256 bit reads for previous and current line but sharing the same DIU interface
	W	Single 256 bit writes.
TE(TD)	R	Single 256 bit reads. Each read returns 2 times 128 bit tags.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering. Dither matrices have start address, end address and line advance increment.
DNC	R	Single 256 bit dead nozzle table reads. Each dead nozzle table read contains 16 dead-nozzle tables entries each of 10 delta bits plus 6 dead nozzle mask bits.
DWU	W	Single 256 bit writes since enable/disable DRAM access per color plane.
LLU	R	Single 256 bit reads since enable/disable DRAM access per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit word can contain 4 PCU commands. PCU reads from DRAM used for reprogramming PEP should be executed with minimum latency. If this occurs between pages then there will be free bandwidth as most of the other SoPEC Units will not be requesting from DRAM. If this occurs between bands then the LDB, CDU and TE bandwidth will be free. So the PCU should have a high priority to access to any spare bandwidth.
Refresh		Single refresh.

20.5   BUFFERING REQUIRED IN SoPEC DRAM REQUESTERS

If each DIU access is a single 256-bit access then we need to provide a 256-bit double buffer in the DRAM requester. If the DRAM requester has a 64-bit interface then this can be implemented as an 8 x 64-bit FIFO.

Table 66. Buffer sizes in SoPEC DRAM requesters

DRAM Requester	Direction	Access patterns	Buffering required in block
CPU	R	Single 256-bit reads.	Cache.
	W	Single 32-bit writes but allowing 16-bit or byte addressable writes.	None.
SCB	W	Single 256-bit writes.	Double 256-bit buffer.
CDU	R	Single 256-bit reads of the compressed contone data.	Double 256-bit buffer.
	W	Each CDU access is a write to 4 consecutive DRAM words in the same row but only 64 bits of each word are written with the remaining bits write masked.	Double half JPEG block buffer.
CFU	R	Single 256 bit reads.	Double 256-bit buffer.
LBD	R	Single 256 bit reads.	Double 256-bit buffer.
SFU	R	Separate single 256 bit reads for previous and current line but sharing the same DIU interface	Double 256-bit buffer for each read channel.
	W	Single 256 bit writes.	Double 256-bit buffer.
TE(TD)	R	Single 256 bit reads.	Double 256-bit buffer.
TE(TFS)	R	Single 256 bit reads. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.	Double line-buffer for 136 bytes implemented in TE.
HCU	R	Single 256 bit reads. 128 x 128 dither matrix requires 4 reads per line with double buffering. 256 x 256 dither matrix requires 8 reads at the end of the line with single buffering.	Configurable between double 128 byte buffer and single 256 byte buffer.
DNC	R	Single 256 bit reads	Double 256-bit buffer. Deeper buffering could be specified to cope with local clusters of dead nozzles.
DWU	W	Single 256 bit writes per enabled odd/even color plane.	Double 256-bit buffer per color plane.
LLU	R	Single 256 bit reads per enabled odd/even color plane.	Double 256-bit buffer per color plane.
PCU	R	Single 256 bit reads. Each PCU command is 64 bits so each 256 bit DRAM read can contain 4 PCU commands. Requested command is read from DRAM together with the next 3 contiguous 64-bits which are cached to avoid unnecessary DRAM reads.	Single 256-bit buffer.
Refresh		Single refresh.	None.

20.6 SoPEC DIU Bandwidth Requirements

Table 67: SoPEC DIU Bandwidth Requirements

Block Name	Direction	Number of cycles between each 256-bit DRAM access to meet peak bandwidth	Peak Bandwidth which must be supplied (bits/cycle)	Average Bandwidth (bits/cycle)	Number of allocated timeslots (based on peak bandwidth)
CPU	R				
	W				
SCB	W	780^2	0.328	0.328	0.5
CDU	R	128 (SF = 4), 288 (SF = 6), 1:1 compression[3]	$32/n^2$ (SF=n), 0.9 (SF = 6), 2 (SF = 4) (1:1 compression)	$32/10 \cdot n^2$ (SF=n), 0.09 (SF = 6), 0.2 (SF = 4) (10:1 compression)[4]	1 (SF=6) 2 (SF=4)
	W	For individual accesses: 16 cycles (SF = 4), 36 cycles (SF = 6), n^2 cycles (SF=n). Will be implemented as a page mode burst of 4 accesses every 64 cycles (SF = 4), 144 (SF =6), $4 \cdot n^2$ (SF =n) cycles[5]	$64/n^2$ (SF=n), 1.8 (SF = 6), 4 (SF = 4)	$32/n^2$ (SF=n), 0.9 (SF = 6), 2 (SF = 4)[6]	2 (SF=6) 4 (SF=4)
CFU	R	32 (SF = 4), 48 (SF = 6)[7]	$32/n$ (SF=n), 5.4 (SF = 6), 8 (SF = 4)	$32/n$ (SF=n), 5.4 (SF = 6), 8 (SF = 4)	5.5 (SF=6) 8 (SF=4)
LBD	R	256 (1:1 compression)[8]	1 (1:1 compression)	0.1 (10:1 compression)[9]	1
SFU	R	128^{10}	2	2	2
	W	256^{11}	1	1	1
TE(TD)	R	252^{12}	1.02	1.02	1.25
TE(TFS)	R	5 reads per line[13]	0.093	0.093	0.25
HCU	R	4 reads per line for 128 x 128 dither matrix[14]	0.074	0.074	0.25
DNC	R	106 (5% dead-nozzles 10-bit delta encoded)[15]	2.4 (clump of dead nozzles)	0.8 (equally spaced dead nozzles)	2.5
DWU	W	6 writes every 256[16]	6	6	6
LLU	R	8 reads every 256[17]	8	6	8
PCU	R	256^{18}	1	1	1
Refresh		100^{19}	2.56	2.56	2.75
TOTAL			SF = 6: 34 SF = 4: 39.5 excluding CPU	SF = 6: 27.5 SF = 4: 31.2 excluding CPU	SF = 6: 35 excluding CPU. SF= 4: 40.5 excluding CPU

Notes:

1: The number of allocated timeslots is based on 64 timeslots each of 1 bit/cycle but broken down to a granularity of 0.25 bit/cycle.

2: 50 Mbit/s is 0.328 bits/cycle or 256 bits every 780 cycles.

3: At 1:1 compression CDU must read a 4 color pixel (32 bits) every SF^2 cycles.

4: At 10:1 average compression CDU must read a 4 color pixel (32 bits) every $10*SF^2$ cycles.

5: 4 color pixel (32 bits) is required, on average, by the CFU every SF^2 (scale factor) cycles.

The time available to write the data is a function of the size of the buffer in DRAM. 1.5 buffering means 4 color pixel (32 bits) must be written every $SF^2 / 2$ (scale factor) cycles. Therefore, at a scale factor of SF, 64 bits are required every SF^2 cycles.

Since 64 valid bits are written per 256-bit write (Figure 104 on page 282) then the DRAM is accessed every SF^2 cycles i.e. at SF4 an access every 16 cycles, at SF6 an access every 36 cycles.

If a page mode burst of 4 accesses is used then each access takes $(3 + 1 + 1 +1)$ equals 6 cycles. This means at SF, a set of 4 back-to-back accesses must occur every $4*SF^2$ cycles. This assumes the page mode select signal is clocked at 320 MHz. CDU timeslots therefore take 6 cycles.

For scale factors lower than 4 double buffering will be used.

6: The average bandwidth 1/2 the peak bandwidth in the case of 1.5 buffering.

7: 4 color pixel (32 bits) read by CFU every SF cycles. At SF4, 32 bits is required every 4 cycles or 256 bits every 32 cycles. At SF6, 32bits every 6 cycles or 256 bits every 48 cycles.

8: At 1:1 compression require 1 bit/cycle or 256 bits every 256 cycles.

9: The average bandwidth required at 10:1 compression is 0.1 bits/cycle.

10: Two separate reads of 1 bit/cycle.

11: Write at 1 bit/cycle.

12: Each tag can be consumed in at most 126 dot cycles and requires 128 bits. This is a maximum rate of 256 bits every 252 cycles.

13: 17 x 64 bit reads per line in PEC1 is 5 x 256 bit reads per line in SoPEC. Double-line buffered storage.

14: 128 bytes read per line is 4 x 256 bit reads per line. Double-line buffered storage.

15: 5% dead nozzles 10-bit delta encoded stored with 6-bit dead nozzle mask requires 0.8 bits/cycle read access or a 256-bit access every 320 cycles. This assumes the dead nozzles are evenly spaced out. In practice dead nozzles are likely to be clumped. Peak bandwidth is estimated as 3 times average bandwidth.

16: 6 bits/cycle requires 6 x 256 bit writes every 256 cycles.

17: 6 bits/160 MHz SoPEC cycle average but will peak at 2 x 6 bits per 106 MHz print head cycle or 8 bits/ SoPEC cycle. The PHI can equalise the DRAM access rate over the line so that the peak rate equals the average rate of 8 bits/ cycle.

18: Assume one 256 read per 256 cycles is sufficient i.e. maximum latency of 256 cycles per access is allowable.

19: As an example assume refresh must occur every 3.2 ms. Refresh occurs row at a time over 5120 rows of 2 parallel 10 Mbit instances. Each refresh takes 3 cycles. This is equivalent to a timeslot every 100 cycles.

20.7   DIU BUS TOPOLOGY

20.7.1   Basic topology

Table 68. SoPEC DIU Requesters

Read	Write	Other
CPU	CPU	Refresh
CDU	SCB	
CFU	CDU	
LBD	SFU	
SFU	DWU	
TE(TD)		
TE(TFS)		
HCU		
DNC		
LLU		
PCU		

Table 68 shows the DIU requesters in SoPEC. There are 11 read requesters and 5 write requesters in SoPEC as compared with 8 read requesters and 4 write requesters in PEC1. Refresh is an additional requester.

In PEC1, the interface between the DIU and the DIU requesters had the following main features:
- separate control and address signals per DIU requester multiplexed in the DIU according to the arbitration scheme,
- separate 64-bit write data bus for each DRAM write requester multiplexed in the DIU,
- common 64-bit read bus from the DIU with separate enables to each DIU read requester.

Timing closure for this bussing scheme was straight-forward in PEC1. This suggests that a similar scheme will also achieve timing closure in SoPEC. SoPEC has 5 more DRAM requesters but it will be in a 0.13 um process with more metal layers and SoPEC will run at approximately the same speed as PEC1.

Using 256-bit busses would match the data width of the embedded DRAM but such large busses may result in an increase in size of the DIU and the entire SoPEC chip. The SoPEC requestors would require double 256-bit wide buffers to match the 256-bit busses. These buffers, which must be implemented in flip-flops, are less area efficient than 8-deep 64-bit wide register arrays which can be used with 64-bit busses. SoPEC will therefore use 64-bit data busses. Use of 256-bit busses would however simplify the DIU implementation as local buffering of 256-bit DRAM data would not be required within the DIU.

20.7.1.1   CPU DRAM access

The CPU is the only DIU requestor for which access latency is critical. All DIU write requesters transfer write data to the DIU using separate point-to-point busses. The CPU will use the cpu_dataout[31:0] bus. CPU reads will not be over the shared 64-bit read bus. Instead, CPU reads will use a separate 256-bit read bus.

20.7.2   Making more efficient use of DRAM bandwidth

The embedded DRAM is 256-bits wide. The 4 cycles it takes to transfer the 256-bits over the 64-bit data busses of SoPEC means that effectively each access will be at least 4 cycles long. It takes only 3 cycles to actually do a 256-bit random DRAM access in the case of IBM DRAM.

20.7.2.1 Common read bus

If we have a common read data bus, as in PEC1, then if we are doing back to back read accesses the next DRAM read cannot start until the read data bus is free. So each DRAM read access can occur only every 4 cycles. This is shown in Figure 67 with the actual DRAM access taking 3 cycles leaving 1 unused cycle per access.
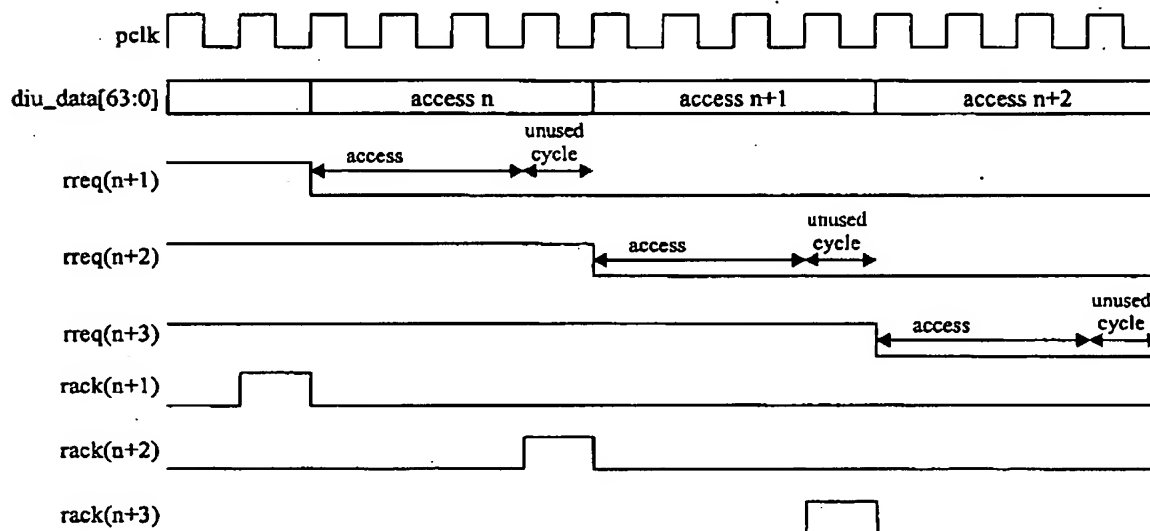

Figure 67. Shared read bus with 3 cycle random DRAM read accesses

20.7.2.2 Interleaving CPU and non-CPU read accesses

The CPU has a separate 256-bit read bus. All other read accesses are 256-bit accesses are over a shared 64-bit read bus. Interleaving CPU and non-CPU read accesses means the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles. Interleaving is achieved by ordering the DIU arbitration slot allocation appropriately.

Figure 68 shows interleaved CPU and non-CPU read accesses.
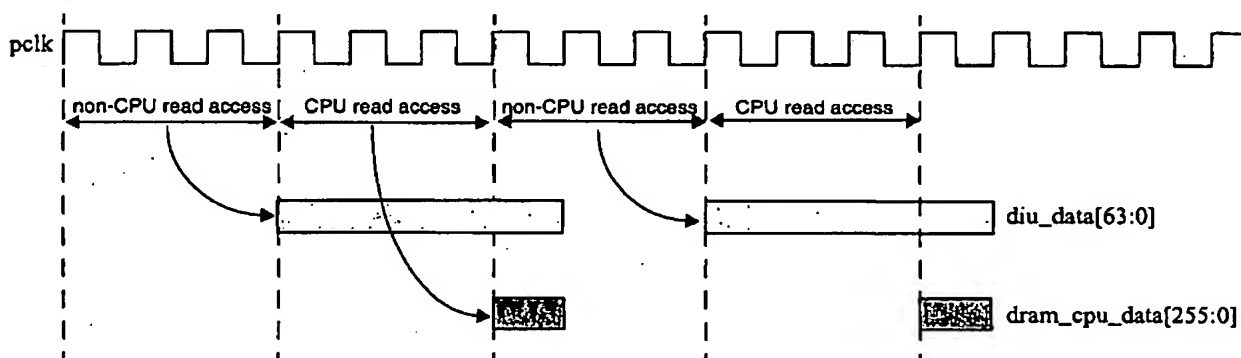
Figure 68. Interleaving CPU and non-CPU read accesses

20.7.2.3 Interleaving read and write accesses

Having separate write data busses means write accesses can be interleaved with each other and with read accesses. So now the effective duration of an interleaved access timeslot is the DRAM access time (3 cycles) rather than 4 cycles. Interleaving is achieved by ordering the DIU arbitration slot allocation appropriately.

Figure 69 shows interleaved read and write accesses. Figure 70 shows interleaved write accesses.
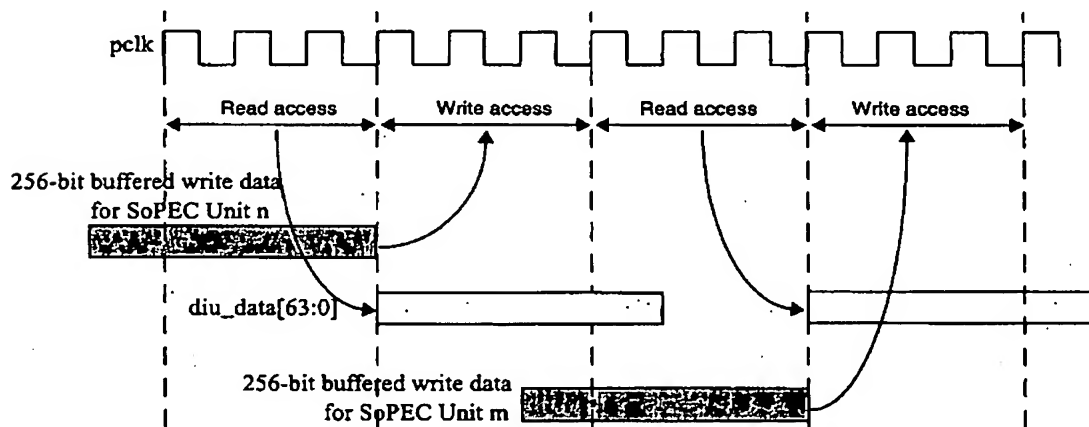
Figure 69. Interleaving read and write accesses with 3 cycle random DRAM accesses

Write data still takes 4 cycles to transmit over 64-bit busses so 256-bit buffers are required in the DIU to gather the write data from the requesters.
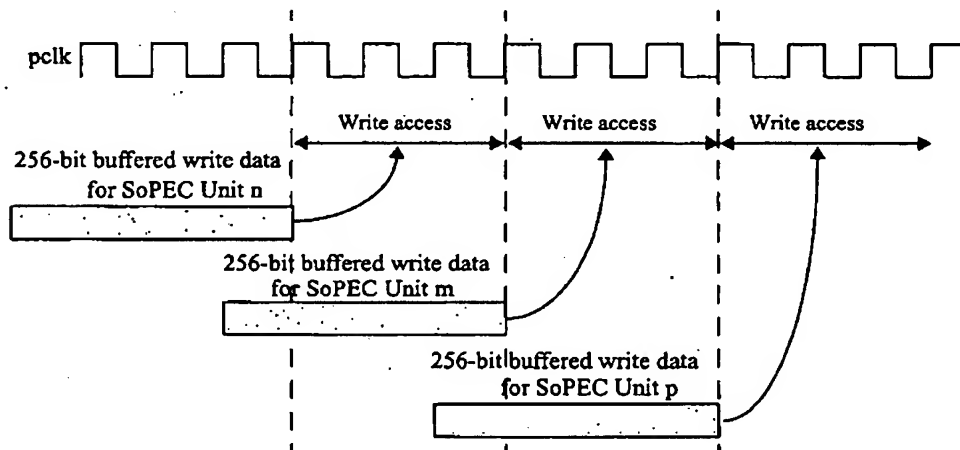
Figure 70. Interleaving write accesses with 3 cycle random DRAM accesses

20.7.3   Bus widths

Table 69. SoPEC DIU Requesters Data Bus Width

Read	Bus access width	Write	Bus access width
CPU	256 (separate)	CPU	32 (OPEN ISSUE)
CDU	64 (shared)	SCB	64
CFU	64 (shared)	CDU	64
LBD	64 (shared)	SFU	64
SFU	64 (shared)	DWU	64
TE(TD)	64 (shared)		
TE(TFS)	64 (shared)		
HCU	64 (shared)		
DNC	64 (shared)		
LLU	64 (shared)		
PCU	64 (shared)		

20.7.4   Conclusions

Reads and writes can be interleaved with a separate 256-bit read bus for the CPU for minimum latency
DIU access. Interleaving can be performed by inserting write accesses or CPU accesses between shared
read bus accesses. The interleaving is achieved by ordering the DIU arbitration slot allocation appropri-
ately.

20.8   SoPEC DRAM ADDRESSING SCHEME

The embedded DRAM is composed of 256-bit words. However the CPU-subsystem may need to write individual bytes of DRAM. Therefore it was decided to make the DIU byte addressable. 22 bits are required to byte address 20 Mbit of DRAM.

Most blocks read or write 256 bit words of DRAM. Therefore only the top 17 bits i.e. bits 21 to 5 are required to address 256-bit word aligned locations.

The exceptions are
- CDU which can write 64-bits so only the top 19 address bits i.e. bits 21-3 are required.
- CPU writes can be 8, 16 or 32-bits. The *cpu_diu_wmask[1:0]* pins indicate whether to write 8, 16 or 32 bits.

All DIU accesses must be within the same 256-bit aligned DRAM word.

20.9   DIU PROTOCOLS

The DIU protocols are
- pipelined i.e the following transaction is initiated while the previous transfer is in progress.
- split transaction i.e. the transaction is split into independent address and data transfers.

20.9.1   Read Protocol except CPU

The SoPEC read requestors, except for the CPU, perform single 256-bit read accesses with the read data being transferred from the DIU in 4 consecutive cycles over a shared 64-bit read bus, *diu_data[63:0]*. The read address *<unit>_diu_radr[21:5]* is 256-bit aligned.

The read protocol is:
- *<unit>_diu_rreq* is asserted along with a valid *<unit>_diu_radr[21:5]*.
- The DIU acknowledges the request with *diu_<unit>_rack*. The request should be deasserted. The minimum number of cycles between *<unit>_diu_rreq* being asserted and the DIU generating an *diu_<unit>_rack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.13.6).
- The read data is returned on *diu_data[63:0]* and its validity is indicated by *diu_<unit>_rvalid*.
- When four *diu_<unit>_rvalid* pulses have been received then if there is a further request *<unit>_diu_rreq* should be asserted again. *diu_<unit>_rvalid* will be always be asserted by the DIU for four consecrative cycles. The first *diu_<unit>_rvalid* pulse will occur 3 cycles after *diu_<unit>_rack* (1 cycle to transfer the address to the DRAM, 2 cycles for the read data to be returned from the DRAM).
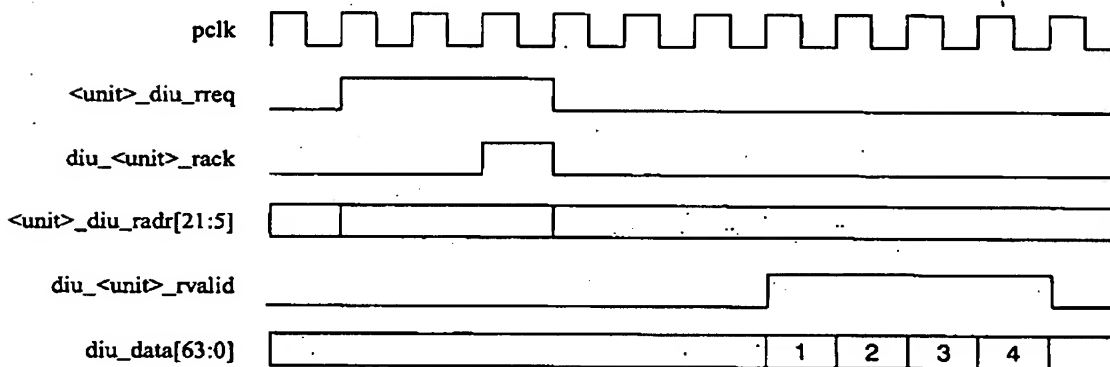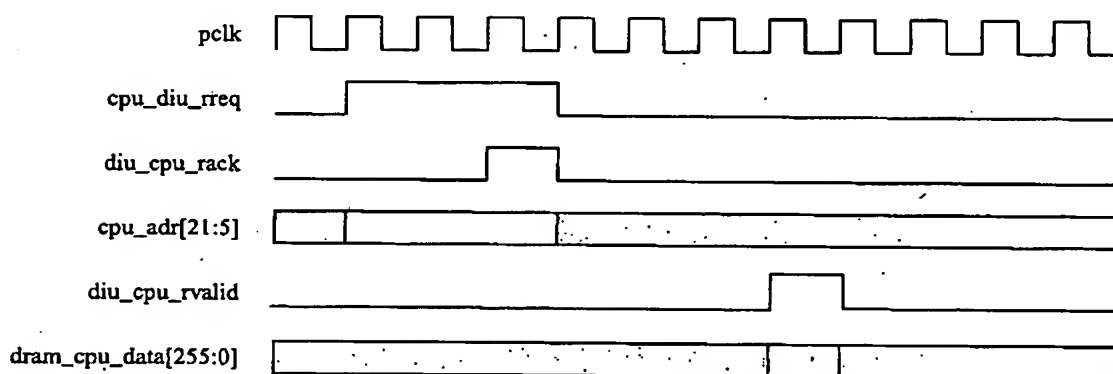

Figure 71. Read protocol for a SoPEC Unit making a single 256-bit access

20.9.2   Read Protocol for CPU

The CPU performs single 256-bit read accesses with the read data being transferred from the DIU over a dedicated 256-bit read bus for DRAM data, *dram_cpu_data[255:0]*. The read address *cpu_adr[21:5]* is 256-bit aligned.

The CPU DIU read protocol is:
- *cpu_diu_rreq* is asserted along with a valid *cpu_adr[21:5]*.

- The DIU acknowledges the request with *diu_cpu_rack*. The request should be deasserted. The minimum number of cycles between *cpu_diu_rreq* being asserted and the DIU generating a *cpu_diu_rack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.13.6).
- The read data is returned on *dram_cpu_data[255:0]* and its validity is indicated by *diu_cpu_rvalid*.
- When the *diu_cpu_rvalid* pulse has been received then if there is a further request *cpu_diu_rreq* should be asserted again. The *diu_cpu_rvalid* pulse will occur 3 cycles after rack (1 cycle to transfer the address to the DRAM, 2 cycles for the read data to be returned from the DRAM).

Figure 72. Read protocol for a CPU making a single 256-bit access

20.9.3 Write Protocol except CPU and CDU

The SoPEC write requestors, except for the CPU and CDU, perform single 256-bit write accesses with the write data being transferred to the DIU in 4 consecutive cycles. over dedicated point-to-point 64-bit write data busses. The write address *<unit>_diu_wadr[21:5]* is 256-bit aligned.

The write protocol is:

- *<unit>_diu_wreq* is asserted along with a valid *<unit>_diu_wadr[21:5]*.
- The DIU acknowledges the request with *diu_<unit>_wack*. The request should be deasserted. The minimum number of cycles between *<unit>_diu_wreq* being asserted and the DIU generating an *diu_<unit>_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.13.6).
- In the clock cycles following *wack* the SoPEC Unit outputs the *<unit>_diu_data[63:0]*, asserting *<unit>_diu_wvalid*. Write data should be output as soon as possible after receiving the *wack*. Accessing registers, register arrays or SRAMs may incur different delays. The first *<unit>_diu_wvalid* pulse can occur in the clock cycle after *diu_<unit>_wack*. In the case of register array or SRAM access, the first *<unit>_diu_wvalid* pulse will occur 2 clock cycles after *diu_<unit>_wack*.
- Once all the write data has been output then if there is a further request *<unit>_diu_wreq* should be asserted again.

A timeout mechanism will be implemented to ensure that the DIU will not lock-up if four *<unit>_diu_wvalid* pulses are not provided.
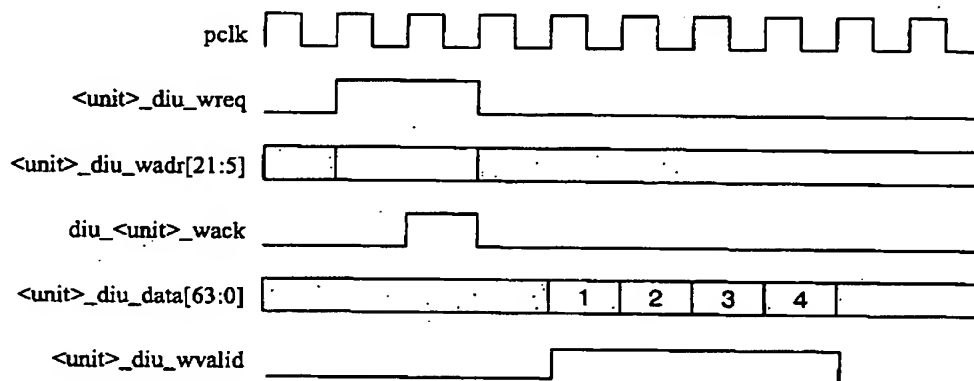
Figure 73. Write Protocol shown for a SoPEC Unit making a single 256-bit access

20.9.4 CPU Write Protocol

The CPU performs single write which can be 8, 16 or 32-bits with the write data being transferred to the DIU over the *cpu_dataout[31:0]* bus. The write address *cpu_adr[21:0]* is byte aligned.

The CPU write protocol is:

- *cpu_diu_wreq* is asserted along with a valid *cpu_adr[21:0]* and a write mask *cpu_diu_wmask[1:0]* to indicate whether an 8, 16 or 32-bit access is required.

- The DIU acknowledges the request with *diu_cpu_wack*. The request should be deasserted. The minimum number of cycles between *cpu_diu_wreq* being asserted and the DIU generating an *diu_cpu_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.13.6).

- In the clock cycle following *diu_cpu_wack* the CPU outputs the *cpu_dataout[31:0]*, asserting *cpu_diu_wvalid*. Write data should be output as soon as possible after receiving the *diu_cpu_wack*. The earliest the *cpu_diu_wvalid* pulse can occur is in the first clock cycle after *diu_cpu_wack*.

- Once the write data has been output then if there is a further request *cpu_diu_wreq* should be asserted again.
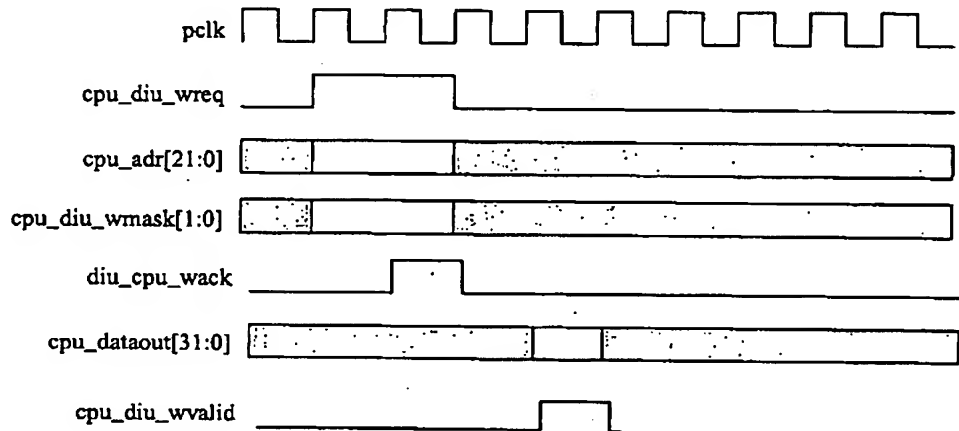
Figure 74. Write Protocol shown for a CPU making an 8, 16 or 32-bit access

20.9.5 CDU Write Protocol

The CPU performs four 64-bit writes to 4 contiguous 256-bit DRAM addresses with the first address specified by *cdu_diu_wadr[21:3]*. The write address *cdu_diu_wadr[21:3]* is 64-bit aligned

The write protocol is:

- *cdu_diu_wdata* is asserted along with a valid *cdu_diu_wadr[21:3]*.
- The DIU acknowledges the request with *diu_cdu_wack*. The request should be deasserted. The minimum number of cycles between *cdu_diu_wreq* being asserted and the DIU generating an *diu_cdu_wack* strobe is 2 cycles (1 cycle to register the request, 1 cycle to perform the arbitration - see Section 20.13.6).
- In the clock cycles following *wack* the CDU outputs the *cdu_diu_data[63:0]*, together with asserted *cdu_diu_wvalid*. Write data should be output as soon as possible after receiving the *wack*. Accessing registers, register arrays or SRAMs may incur different delays. The first *cdu_diu_wvalid* pulse can occur in the clock cycle after *diu_cdu_wack*. In the case of register array or SRAM access, the first *cdu_diu_wvalid* pulse will occur 2 clock cycles after *diu_cdu_wack*.
- Once all the write data has been output then if there is a further request *cdu_diu_wreq* should be asserted again.

A timeout mechanism will be implemented to ensure that the DIU will not lock-up if four *cpu_diu_wvalid* pulses are not provided.
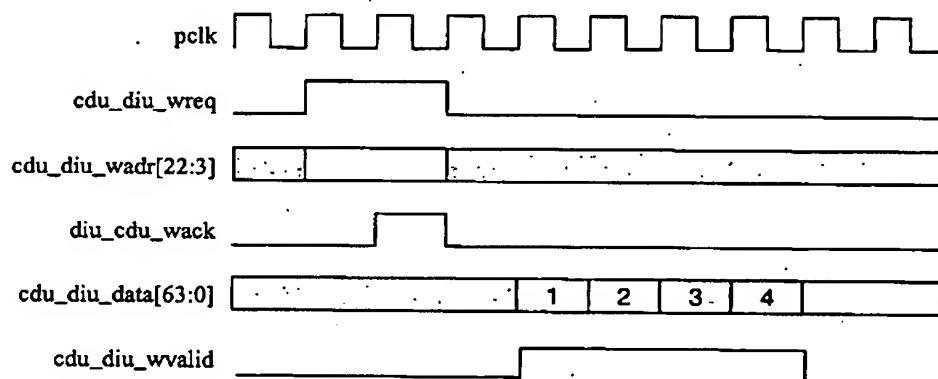
Figure 75. Write Protocol shown for CDU making four contiguous 64-bit accesses

20.10   DIU ARBITRATION MECHANISM

The DIU will arbitrate access to the embedded DRAM. The arbitration scheme is outlined in the next sections.

20.10.1   Timeslot based arbitration scheme

Table 67 summarised the bandwidth requirements of the SoPEC requestors to DRAM. If we allocate the DIU requestors in terms of peak bandwidth then we require 36 bits/cycle (at SF =6) and 42.5 bits/cycle (at SF = 4) for all the requestors except the CPU.

A timeslot scheme is defined with 64 *main* timeslots. The number of used main timeslots is programmable between 0 and 64.

Since DRAM read requestors, except for the CPU, are connected to the DIU via a 64-bit data bus each 256-bit DRAM access requires 4 *pclk* cycles to transfer the read data over the shared read bus. The timeslot rotation period for 64 timeslots each of 4 *pclk* cycles is 256 *pclk* cycles or 1.6 μs, assuming *pclk* is 160 MHz. Each timeslot represents a 256-bit access every 256 *pclk* cycles or 1 bit/cycle. This is the granularity of the majority of DIU requestors bandwidth requirements in Table 67.

The SoPEC DIU requesters can be represented using 5 bits (Table on page 229). Using 64 timeslots means that to allocate each timeslot to a requester a total of 64 times 5 configuration registers is required for the 64 main timeslots.

Timeslot based arbitration works by having a pointer point to the current timeslot. When re-arbitration.is signaled the arbitration pointer will advance to the next timeslot. If the SoPEC Unit assigned to the current timeslot is not requesting then the unused timeslot arbitration mechanism outlined in Section 20.10.4 is used to select the arbitration winner.

The timeslot pointer advances when the DIU issues the next command to the DRAM. Each timeslot therefore denotes a single access. The duration of the timeslot depends on the access.

If the SoPEC Unit pointed to by the current timeslot pointer is not requesting then the slot will be allocated according to the mechanism described in Section 20.10.5.
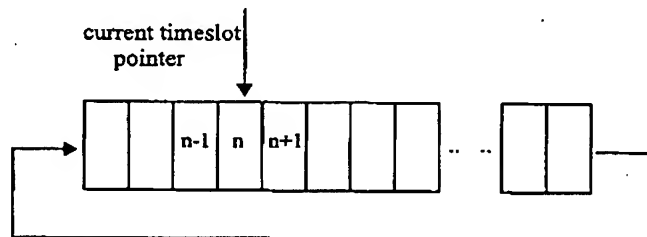

Figure 76.   Timeslot based arbitration

20.10.2   Separate read and write arbitration windows

For write accesses, except the CPU, 256-bits of write data are transferred from the SoPEC DIU write requestors over 64-bit write busses in 4 clock cycles. This write data transfer latency means that writes accesses, except for CPU writes, must be arbitrated 4 cycles in advance. The [to be included figure and explanation] shows why this is necessary.

Since write arbitration must occur 4 cycles in advance, and the minimum duration of a timeslot duration is 3 cycles, the arbitration rules must be modified to initiate write accesses in advance.accordingly. There is a timeslot lookahead pointer shown in Figure 77 two timeslots in advance of the current timeslot pointer. .
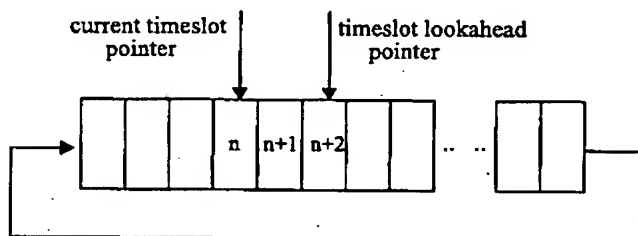

Figure 77. Timeslot based arbitration with separate read and write pointers

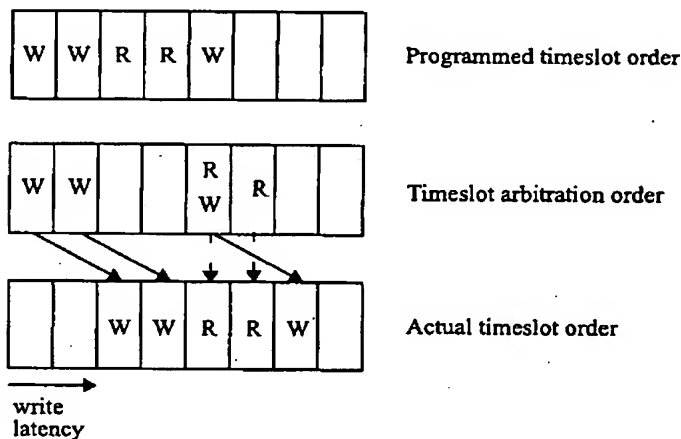The following examples illustrate separate read and write timeslot arbitration.

Figure 78. Example (a), separate read and write arbitration

In Figure 78 writes are arbitrated two timeslots in advance. Reads are arbitrated in the same cycle Writes can be arbitrated in the same cycle as a read. During arbitration the command address of the arbitrated SoPEC Unit is captured.

Other examples are shown in Figure 79, Figure 80 and Figure 81. The actual timelsot order is always the same as the programmed timeslot order i.e. out of order accesses do not occur and data coherency is never an issue.

Each write must always incur a latency of two timeslots. If the first write occurs in the first timeslot then all following timeslots will incur a latency of two timeslots. This is shown in Figure 78 and Figure 79. If the first write occurs in the second timeslots then all following timeslots will incur a latency of two timeslots. This is shown in Figure 80.
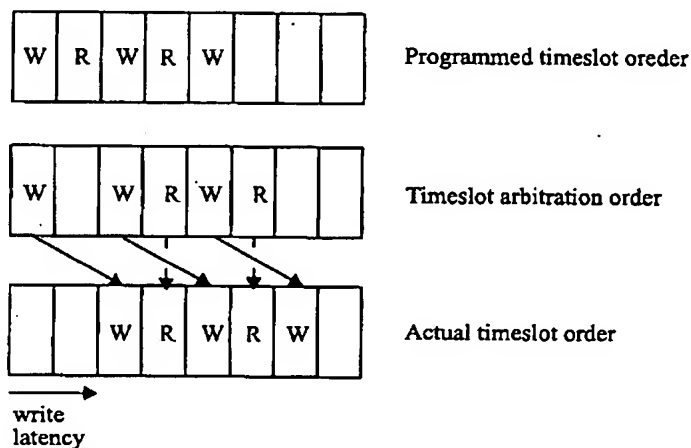
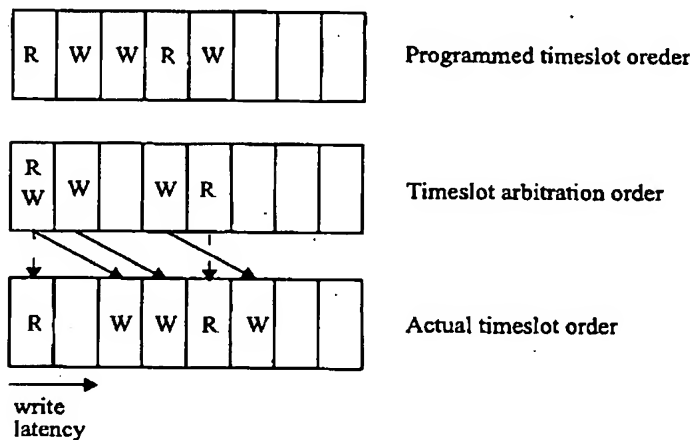Figure 79. Example (b), separate read and write arbitration

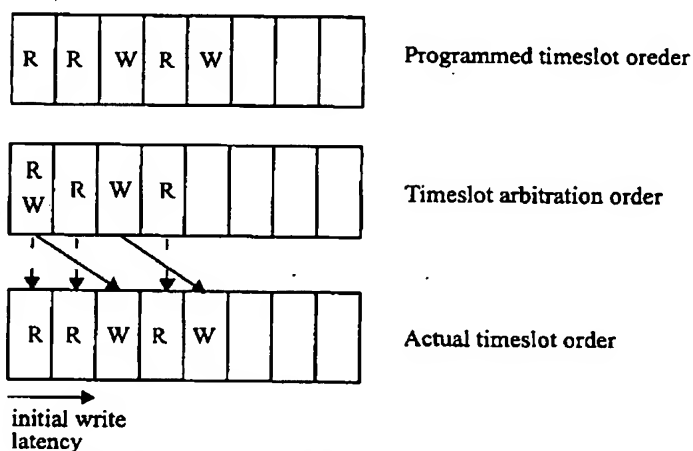Figure 80. Example (c), separate read and write arbitration

Programmed timeslot oreder

Timeslot arbitration order

Actual timeslot order

initial write
latency

Figure 81. Example (d), separate read and write arbitration

Table 70 shows the 4 scenarios depending on whether the current timeslot and timeslot lookahead pointers point to read or write accesses.

To be checked and updated:

Table 70: Arbitration with separate windows for read and write accesses

current timeslot pointer	timeslot lookahead pointer	action
read	write	Initiate read transfer.
		Initiate write transfer.
read1	read2	Initiate read1 transfer.
write1	write2	Initiate write2 transfer.
write	read	No action.

If the current timeslot pointer points to a read access then this will be initiated immediately.

If the timeslot lookahead pointer points to a write access then this access is initiated immediately, or immediately after the read access associated with the current timeslot pointer is initiated.

When a write access is initiated the DIU will capture the write address and will do the DRAM write two tiemslots in advance when the associated write data has been transfered to the DIU.

To be checked and updated: At initialisation, both pointers point to the first timeslot. The lookahead pointer advances to the second timeslot and the third timeslot in successive clock cycles until it is two timeslots ahead of the current timeslot pointer. Then both pointers advance in tandem. At each step, the rules in Table 70 are obeyed. This leads to the behaviour shown in the exampes of Figure 78 to Figure 81.

CPU write accesses are excepted from the lookahead mechanism.

Timing diagrams for these scenarios are shown in Section 20.13 Implementation.

If the selected SoPEC Unit is not requesting then there will be separate read and write selection for unused timeslots. This is described in Section 20.10.5.

20.10.3 Arbitration of CPU accesses

The CPU can be allocated timeslots like any other DIU requestor. If CPU accesses are interleaved between the shared read bus accesses then the DIU timeslots will take 3 cycles as shown in Section 20.7.2.2. The timeslot rotation will be faster than 256 *pclk* cycles.

What distinguishes the CPU from other SoPEC requestors, is that the CPU requires minimum latency DRAM access i.e. preferably the CPU should get the next available timeslot whenever it requests.

The minimum CPU read access latency is estimated in Table 71. This is the time between the CPU making a request to the DIU and receiving the read data back from the DIU. This ignores any latency associated with the CPU's caching mechanism.

Table 71. Estimated CPU read access latency ignoring caching

CPU read access latency	Duration
register the CPU read request	1 cycle
complete the arbitration of the request	1 cycle
transfer the read address to the DRAM	1 cycle
DRAM read latency	2 cycles
register the read data	1 cycle
TOTAL	6 cycles

If the CPU, as is likely, requests DRAM access again immediately after receiving data from the DIU then the CPU can access every second timeslot. This assumes that interleaving is employed so that timeslots last 3 cycles. If the CPU access latency increases to 7 cycles then the CPU will only be able to access every third timeslot.

If a cache hit occurs the CPU does not require DRAM access. For its next DIU access it will have to wait for its next assigned DIU slot. Cache hits therefore will reduce the number of DRAM accesses but not speed up any of those accesses.

To avoid the CPU having to wait for its next timeslot it is desirable to have a mechanism for ensuring that the CPU always gets the next available timeslot without incurring any latency on the non-CPU timeslots.

This can be done by defining each timeslot as consisting of a CPU access preceding a non-CPU access. Each timeslot will last 6 cycles i.e. a CPU access of 3 cycles and a non-CPU access of 3 cycles. This is exactly the interleaving behaviour outlined in Section 20.7.2.2. If the CPU does not require an access, the timeslot will take 3 or 4 and the timeslot rotation will go faster. A summary is given in Table 72.

Table 72. Timeslot access times.

CPU Access	Duration	Explanation
CPU access + non-CPU access	3 + 3 = 6 cycles	Interleaved access
non-CPU access	4 cycles	Access and preceding access both to shared read bus

Table 72. Timeslot access times.

Access	Duration	Explanation
non-CPU access	3 cycles	Access and preceding access not both to shared read bus
CDU write access	3+1+1+1 = 6 cycles	Page mode select signal is clocked at 320 MHz

CDU write accesses require 6 cycles. CDU write accesses preceded by a CPU access require 9 cycles. CDU timeslots therefore take longer than all other DIU requestors timeslots.

With a 256 cycle rotation there can be 42 accesses of 6 cycles. This is just enough timeslots for SF = 4 operation, ignoring implementation pipeline latencies.

For low scale factor applications, it is desirable to have more timeslots available in the same 256 cycle rotation. So two counters of 4-bits each are defined allowing the CPU to get a maximum of *cpu_timeslots* in *total_timeslots*. A timeslot counter starts at *total_timeslots* and decrements every timeslot, while another counter starts at *cpu_timeslots* and decrements every timeslot in which the CPU uses its access. When the CPU timeslot counter goes to zero before *total_timeslots* no further CPU accesses are allowed. When the *total_timeslots* counter reaches zero both counters are reset to their respective initial values.

When *cpu_timeslots* is set to zero then no accesses will be preceded by CPU accesses. The CPU can be allocated timeslots like any other DIU requestor.

If CPU accesses are interleaved between the shared read bus accesses then the DIU timeslots will take 3 cycles as shown in Section 20.7.2.2 Otherwise the timeslots will take 4 cycles each and the rotation will take 256 cycles.

The various modes of operation are summarised in Table 73 with a nominal rotation period of 256 cycles.

Table 73. CPU timeslot allocation modes with nominal rotation period of 256 cycles

Access Type	Nominal Timeslot duration	Number of timeslots	Notes
CPU Pre-access i.e. *cpu_timeslots = total_timeslots*	6 cycles	42 timeslots	Each access is CPU + non-CPU. If CPU does not use a timeslot then rotation is faster.
Fractional CPU Pre-access i.e. *cpu_timeslots < total_timeslots*	4 or 6 cycles	42-64 timeslots	Each CPU + non-CPU access requires a 6 cycle timeslot.
			Individual non-CPU timeslots take 4 cycles if current access and preceding access are both to shared read bus.
			Individual non-CPU timeslots take 3 cycles if current access and preceding access are both to shared read bus.
Interleaved i.e. *cpu_timeslots = 0*	4 cycles	64 timeslots	Timeslot rotation is faster by 1 cycle for each CPU, write access or interleaved read access

20.10.4 Sub-timeslots

Looking at the bandwidth requirements of the DIU requesters in Table 67, most DIU requesters require bandwidths of 1 bit/cycle or multiples thereof. However, some of the requestors require much lower band-

width. This suggests that some sub-timeslots of lower granularity than a nominal 1 bit/cycle should be defined.

There will be 2 sub-timeslots of 4 and 8 slots each. The bandwidth associated with each individual sub-slot is nominally 0.25 and 0.125 bits/cycle respectively, assuming each slot last 4 cycles. Sub-timeslots can be allocated to any number of main timeslots so that any multiple of the individual sub-timeslot bandwidth can be obtained.

Table 74. Sub-timeslot definition

Sub-timeslot	Number of slots	Bandwidth of each slot (assuming 4 cycles)
Sub4timeslot	4	0.25 bits/cycle
Sub8timeslot	8	0.125 bits/cycle

Each sub-slot pointer gets advanced each time it is accessed regardless if it slot is used or not.

Sub-timeslots are similar in all other ways to main timeslots i.e.
- they can have preceding CPU accesses in a similar manner.
- unused slots are decided by the same unused timeslot allocation mechanism (Section 20.10.5).
- a timeslot lookahead pointer is used to select writes (except for CPU writes) early to compensate for write data transfer latency.
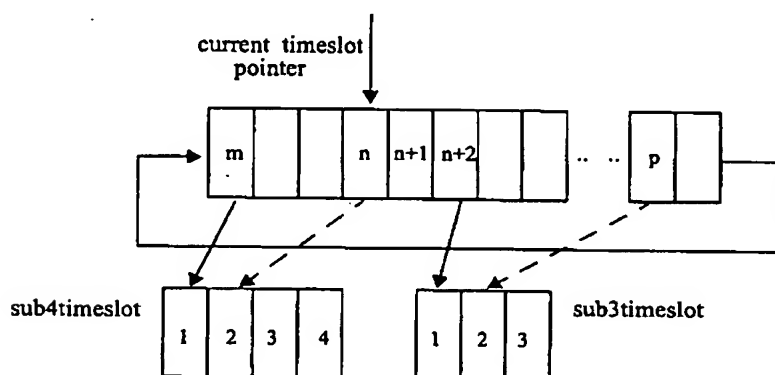

Figure 82. Example sub-timeslot allocation

An example sub-timeslot allocation is shown in Figure 82.

Every time main timeslots *m* and *n* are accessed, the SoPEC unit pointed to by the pointer in *sub4timeslot* will win arbitration and the *sub4timeslot* pointer will advance. Similarly, every time main timeslots *n+2* and *p* are accessed, the SoPEC unit pointed to by the pointer in *sub3timeslot* will win arbitration and the *sub3timeslot* pointer will advance.

20.10.5 Allocating unused timeslots

Unused slots are re-allocated on a two-level round-robin basis. This is best-effort traffic.

Each SoPEC requestor has two associated bits, *RoundRobinLevel* indicates whether it is in level 1 or level 2 round-robin, and *RoundRobinEnable* indicates whether it is enabled or not in the selected round-robin.

Table 75. Round-robin selection

Level	Enable	Action
RoundRobinLevel = 0	RoundRobinEnable = 0	Not enabled
	RoundRobinEnable = 1	Level 1
RoundRobinLevel = 1	RoundRobinEnable = 0	Not enabled
	RoundRobinEnable = 1	Level 2

Separate read and write round-robin trees are needed, one for read accesses and one for write accesses.

CDU write accesses cannot be included in the round-robin allocation for write as CDU accesses take 6 cycles. The write accesses which the CDU write could otherwise replace require only 3 or 4 cycles.

Robin-robin allocations do not have CPU pre-accesses.

A pointer points to the current allocated unit in each of the round-robin levels. If the unit pointed to the level 1 round-robin is requesting then this unit wins the arbitration and the pointer is advanced. If the unit pointed to in the level 1 round-robin is not requesting then the next units in the level 1 round-robin are examined. When a requesting unit is found this unit wins the arbitration and the pointer advances to the next unit. If no unit is requesting then the pointer does not advance and the second level of round-robin is examined in the same way as first level of the round-robin.

Table 76. Write round-robin registers bit order

Name	Bit Index
CPU(W)	0
SCB	1
SFU(W)	2
DWU	3

20.10.6 Background refresh controller

A background refresh controller should be implemented that will issue a refresh and pause the timeslot rotator in case data is about to be lost. This scenario will only occur in the situation that insufficient timeslots were allocated for refresh.

20.11 GUIDELINES FOR PROGRAMMING THE DIU

Some guidelines for programming the DIU arbitration scheme are given in this section together with an example.

20.11.1 Implementation pipeline latencies

The number of allocated timeslots for each requester needs to take into account implementation pipeline latencies. The number of timeslots is made programmable. This means 1 or 2 timeslots can be removed to allow for implementation latency. Each timeslot will allow for 6 cycles implementation latency in *CPU Pre-access* mode and 3 cycles otherwise for each single timeslot allocation in a rotation.. If units are allocated more than 1 timeslot in a rotation then the gap between slots may need to be reduced additionally to allow for implementation latency.

20.11.2 Ensuring sufficient DNC and PCU access

PCU command reads from DRAM are exceptional events and should complete in as short a time as possible. Similarly, we must ensure there is sufficient free bandwidth for DNC accesses e.g. when clusters of dead nozzles occur. In Table 67 DNC is allocated 3 times average bandwidth. PCU and DNC can also be allocated to the level 1 round-robin allocation for unused timeslots so that unused timeslot bandwidth is available to them.

20.11.3 Basing timeslot allocation on peak bandwidths

Since the embedded DRAM provides sufficient bandwidth to use 1:1 compression rates for the CDU and LBD, it is possible to simplify the main timeslot and sub-timeslot allocation by basing the allocation on peak bandwidths. The only variable in determining timeslot allocations then becomes the scale factor.

If slot allocation is based on peak bandwidth requirements then DRAM access will be *guaranteed* to all SoPEC requesters. If we do not allocate slots for peak bandwidth requirements then we can also allow for the peaks *deterministically* by adding some cycles to the print line time.

20.11.4 Adjacent timeslot limitations

All DIU requesters have state-machines which request and transfer the read or write data before requesting again. The time to perform this operation is greater than the time between adjacent timeslots. Therefore adjacent timeslots should not be assigned to a particular DIU requester because the requester will not be able to make use of all these slots.

20.11.5 Line margin

The SFU must output 1 bit/cycle to the HCU. Since *HCUNumDots* may not be a multiple of 256 bits the last 256-bit DRAM word on the line can contain extra zeros. In this case, the SFU may not be able to provide 1 bit/cycle to the HCU. This could lead to a stall by the SFU. This stall could then propagate if the margins being used by the HCU are not sufficient to hide it. The maximum stall can be estimated by the calculation: DRAM service period - X scale factor * dots used from last DRAM read for HCU line.

Similarly, if the line length is not a multiple of 256-bits then e.g. the LLU could read data from DRAM which contains padded zeros. This could lead to a stall. This stall could then propagate if the page margins cannot hide it.

A single addition of 256 cycles to the line length will suffice for all DIU requesters to mask these stalls.

20.11.6 Example DIU programming

A full example to be worked out.

Program *MainTimeslot* and *SubnTimeslot* configuration registers (Table 82) for peak required bandwidths of SoPEC Units according to the scale factor used for the document.

Program unused slots to use the round-robin allocation to share unused slots between all DIU requesters.

20.12 CPU DRAM ACCESS PERFORMANCE

This section does not yet reflect any implementation pipeline latencies.

The CPU's share of the timeslots can be specified in terms of guaranteed bandwidth and average bandwidth allocations.

The CPU's access rate to memory depends on
- the CPU read access latency i.e. the time between the CPU making a request to the DIU and receiving the read data back from the DIU.
- how often it can get access to DIU timeslots.

Table 71 estimated the CPU read latency ignoring caching as 6 cycles.

How often the CPU can get access to DIU timeslots depends on the access type.

Table 77. CPU DRAM access performance

Access type	Nominal Timeslot duration	CPU DRAM access rate	Notes
CPU Pre-access	6 cycles	Lower bound (guaranteed bandwidth) is 160 MHz / 6 = 26.27 MHz	CPU can access every timeslot
Fractional CPU Pre-access	6 cycles	Lower bound (guaranteed bandwidth) is (160 MHz * N / P)	CPU accesses precede a fraction N of timeslots where N = C/T. C = *cpu_timeslots* T = *total_timeslots* P = (6*C + 4*(T-C)) / T
Interleaved	4 cycles	See Section 20.12.1	At SF = 6, 28 timeslots available for CPU. At SF = 4, 21 timeslots available for CPU.

For *CPU Pre-access* and *Fractional CPU Pre-access* modes average and guranteed CPU bandwidth are equivalent since the CPU is limited to a certain fraction of timeslots.

If the CPU runs out of its instruction cache then instruction fetch performance is only limited by the on-chip bus protocol. With a 2 cycle bus protocol (address cycle + data cycle) the performance would be 80 MHz.

20.12.1 CPU DRAM access performance with interleaved access mode

Table 78 shows the *guaranteed* periodic CPU access with 4 cycle DRAM access and *pclk* = 160 MHz.

Table 78. Guaranteed Periodic CPU access with 4 cycle timeslots and *pclk* = 160 MHz

Guaranteed Periodic	SF=6	SF=4
Timeslots left for CPU	28.25	21.5
Maximum wait for timeslot	12 cycles	12 cycles
CPU rate	13.3 MHz	13.3 MHz

Since timeslots are integral multiples of 4 cycles the maximum wait for a timeslot and hence minimum the CPU rate must reflect this.

Table 79 shows the *average* CPU access with 4 cycle DRAM access and *pclk* = 160 MHz. This will be a bursty access.

Table 79. Average bursty CPU access with 4 cycle DRAM access and *pclk* = 160 MHz

Average (bursty)	SF=6	SF=4
Timeslots left for CPU	34.95	30.8
Maximum wait for timeslot	8 cycles	12 cycles
CPU rate	20 MHz	13.3 MHz

Interleaving of CPU and write accesses with shared read bus accesses will mean some of the timeslots will take 3 cycles rather than 4 cycles. This will mean that CPU slots will be available more frequently and higher CPU performance is attainable.

20.13 IMPLEMENTATION

The DRAM Interface Unit (DIU) is partitioned into 2 logical blocks to facilitate design and verification.

a. The DRAM Access Unit (DAU) which interfaces to the SoPEC DIU requesters.

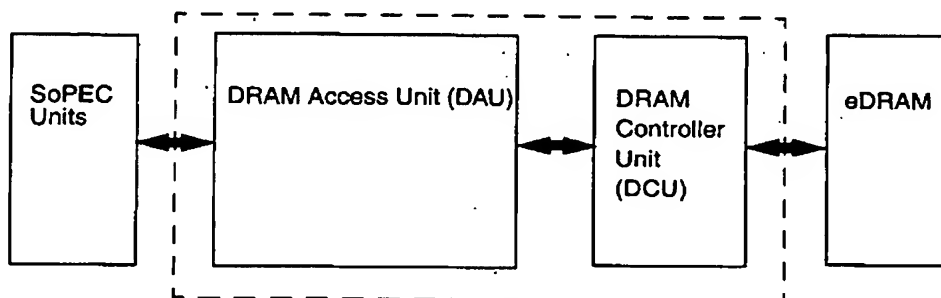b. The DRAM Controller Unit (DCU) which accesses the embedded DRAM.

Figure 83. DIU Partition

The basic principle in design of the DIU is to ensure that the eDRAM is accessed at its maximum rate while keeping the circuit latency for each access as low as possible.

The DCU is designed to interface with single bank 20 Mbit IBM Cu-11 embedded DRAM performing random accesses every 3 cycles. Page mode write accesses, associated with the CDU, are also supported.

The DAU is designed to support interleaved accesses allowing the DRAM to be accessed every 3 cycles where back-to-back accesses do not occur over the shared 64-bit read data bus.

20.13.1 Definition of DCU IO

Table 80. DCU Interface

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock
prst_n	1	In	Active-low, synchronous reset in pclk domain
Inputs from DAU			
dau_dcu_cmdavail	1	In	Signal indicating a DAU command is available i.e. *dau_cmd_adr, dau_cmd_rwn* and *dau_cmd_refresh* are valid.
dau_dcu_cmdadr[21:5]	17	In	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_cmdrwn	1	In	Signal indicating the direction for the DRAM access (1=read, 0=write).
dau_dcu_cmdrefresh	1	In	Signal indicating that a refresh command is to be issued. If asserted *dau_cmd_adr* and *dau_cmd_rwn* will be ignored.
dau_dcu_wdata	256	In	256-bit write data to DCU
dau_dcu_wmask	256	In	256-bit write data mask to DCU
dau_dcu_wvalid	17	In	Signal indicating valid write data and write mask.
Outputs to DAU			
dcu_dau_cmdaccept	1	Out	Signal indicating that the DCU has accepted a valid command from the DAU.
dcu_dau_refreshcomplete	1	Out	Signal indicating that the DCU has completed a refresh.
dcu_dau_rdata	256	Out	256-bit read data from DCU.
dcu_dau_rrvalid	1	Out	Signal indicating valid read data on *dcu_rdata*.
Outputs to DRAM			
~			
Inputs from DRAM			

20.13.2 Definition of DAU IO

Table 81. DAU Interface

Port Name	# Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock
prst_n	1	In	Active-low, synchronous reset in pclk domain
CPU Interface			
cpu_adr[9:2]	8	In	CPU address bus. 8 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
diu_cpu_data[31:0]	32	Out	Configuration, status and debug read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU access code signals. cpu_acode[0] - Program (0) / Data (1) access cpu_acode[1] - User (0) / Supervisor (1) access The DAU will only allow supervisor mode accesses to data space.
cpu_diu_sel	1	In	Block select from the CPU. When *cpu_diu_sel* is high both *cpu_addr* and *cpu_dataout* are valid
diu_cpu_rdy	1	Out	Ready signal to the CPU. When *diu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *diu_cpu_data* is valid.
diu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
DIU Read Interface to SoPEC Units			
<unit>_diu_rreq	1	In	SoPEC unit requests DRAM read. A read request must be accompanied by a valid read address.
<unit>_diu_radr[21:5]	17	In	Read address to DIU 17 bits wide (256-bit aligned word).
diu_<unit>_rack	1	Out	Acknowledge from DIU that read request has been accepted and new read address can be placed on *<unit>_diu_radr*
diu_data[63:0]	64	Out	Data from DIU to SoPEC Units except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
dram_cpu_data[255:0]	256	Out	256-bit data from DRAM to CPU.
diu_<unit>_rvalid	1.	Out	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus
DIU Write Interface to SoPEC Units			
<unit>_diu_wreq	1	In	SoPEC unit requests DRAM write. A write request must be accompanied by a valid write address.
<unit>_diu_wadr[21:5]	17	In	Write address to DIU except CPU, CDU 17 bits wide (256-bit aligned word)
cpu_adr[21:0]	22	In	CPU Write address to DIU 22 bits wide (8-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.

Table 81. DAU Interface

Port Name	Pins	I/O	Description
cpu_diu_wmask[1:0]	2	In	Flag indicating format of CPU write to DRAM 00: 8-bit write 01: 16-bit write 10: 32-bit write 11: reserved cpu_addr[2:0] are driven in accordance with the width of the data access indicated by cpu_diu_wmask. Addresses cannot cross a 256-bit word DRAM boundary.
cdu_diu_wadr[21:3]	19	In	CDU Write address to DIU 19 bits wide (64-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
diu_<unit>_wack	1	Out	Acknowledge from DIU that write request has been accepted and new write address can be placed on <unit>_diu_wadr
<unit>_diu_data[63:0]	64	In	Data from SoPEC Unit to DIU except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
cpu_dataout[31:0]	32	In	Data from CPU to DIU.
<unit>_diu_wvalid	1	In	Signal from SoPEC Unit indicating that data on <unit>_diu_data is valid.
Outputs to DCU			
dau_dcu_cmdavail	1	Out	Signal indicating a DAU command is available i.e. dau_cmd_adr, dau_cmd_rwn and dau_cmd_refresh are valid.
dau_dcu_cmdadr[21:5]	17	Out	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_dcu_cmdrwn	1	Out	Signal indicating the direction for the DRAM access (1=read, 0=write).
dau_dcu_cmdrefresh	1	Out	Signal indicating that a refresh command is to be issued. If asserted dau_cmd_adr and dau_cmd_rwn will be ignored.
dau_dcu_wdata	256	Out	256-bit write data to DCU
dau_dcu_wmask	256	Out	256-bit write data mask to DCU
dau_dcu_wvalid	17	Out	Signal indicating valid write data and write mask.
Inputs from DCU			
dcu_dau_cmdaccept	1	In	Signal indicating that the DCU has accepted a valid command from the DAU.
dcu_dau_refreshcomplete	1	In	Signal indicating that the DCU has completed a refresh.
dcu_dau_rdata	256	In	256-bit read data from DCU.
dcu_dau_rvalid	1	In	Signal indicating valid read data on dcu_rdata.

The CPU subsystem bus interface is described in more detail in Section 11.4.3. The DAU block will only allow supervisor mode accesses to data space (i.e. cpu_acode[1:0] = b11). All other accesses will result in diu_cpu_berr being asserted.

20.13.3 DAU Configuration Registers

Table 82. DAU configuration registers

Address (DIU base +)	Register	#bits	Reset	Description
0x00	Reset	1	0x1	A write to this register causes a reset of the DIU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	RefreshPeriod	10	0x000	Background refresh controller. When set to 0 background refresh is off, otherwise value indicates number of cycles between each refresh.
0x08	NumMainTimeslots	7	0x40	Number of main timeslots (0-64)
0x09	CPUTimeslots	4	0x0	CPUTimeslots out of TotalTimeslots are available for the CPU.
0x0A	TotalTimeslots	4	0x0	CPUTimeslots out of TotalTimeslots are available for the CPU.
0x100-0x1FC	MainTimeslot	[64][5]	0x00	Programmable main timeslots (up to 64 main timeslots)
0x200-0x208	Sub3Timeslot	[3][5]	0x00	Programmable sub- timeslots (3 timeslots timeslots)
0x210-0x21C	Sub4Timeslot	[4][5]	0x00	Programmable sub- timeslots (4 timeslots timeslots)
0x220-0x234	Sub6Timeslot	[6][5]	0x00	Programmable sub- timeslots (6 timeslots timeslots)
0x300	ReadRoundRobinLevel	12	0x000	For each read requester plus refresh 0 = level1 of round-robin 1 = level2 of round-robin

Each main timeslot and sub-timeslot can be assigned a SoPEC DIU requestor according to Table 83. Main timeslots can be assigned SoPEC units, refresh and sub-timeslots. Sub-timeslots can be assigned SoPEC units and refresh but not to other sub-timeslots.

Table 83. SoPEC DIU Encoding

Name	Index (binary)	Index (HEX)
None	b00000	0x00
Write		
CPU(W)	b00001	0x01
SCB	b00010	0x02
CDU(W)	b00011	0x03
SFU(W)	b00100	0x04
DWU	b00101	0x05
Read		
CPU(R)	b00110	0x06

Table 83. SoPEC DIU Encoding

Name	Index (binary)	Index (HEX)
CDU(R)	b00111	0x07
CFU	b01000	0x08
LBD	b01001	0x09
SFU(R)	b01010	0x0A
TE(TD)	b01011	0x0B
TE(TFS)	b01100	0x0C
HCU	b01101	0x0D
DNC	b01110	0x0E
LLU	b01111	0x0F
PCU	b10000	0x10
Others		
Refresh	b10001	0x11
Sub1timeslot	b10010	0x12
Sub2timeslot	b10011	0x13
Sub3timeslot	b10100	0x14

ReadRoundRobinLevel and *ReadRoundRobinEnable* registers are encoded in the bit order defined in Table 84.

Table 84. Read round-robin registers bit order

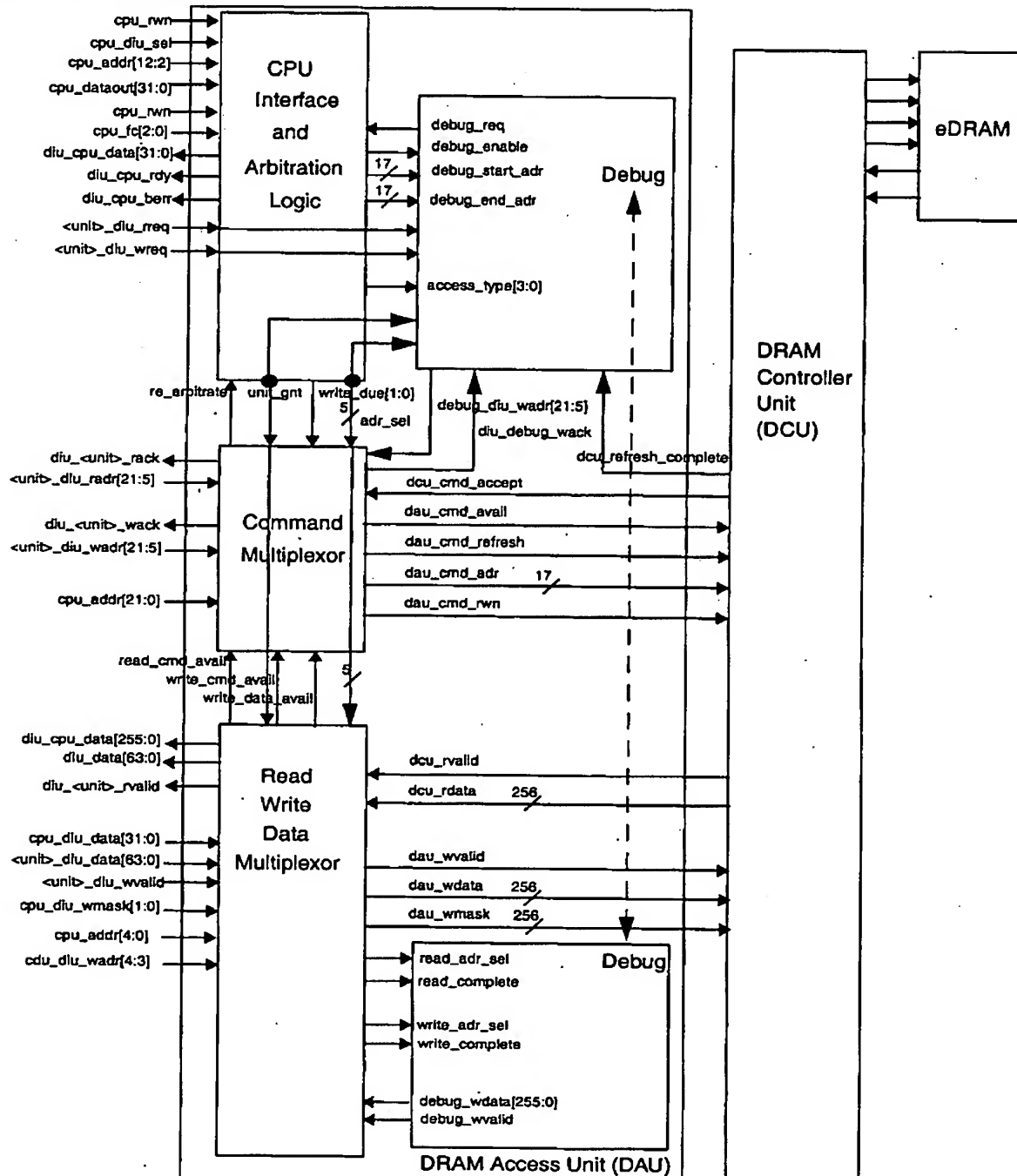| Name	Bit Index
CPU(R)	0
CDU(R)	1
CFU	2
LBD	3
SFU(R)	4
TE(TD)	5
TE(TFS)	6
HCU	7
DNC	8
LLU	9
PCU	10
Refresh	11

20.13.4 DIU Partition

Figure 84. DIU Partition

20.13.5 CPU Interface and Arbitration Logic Sub-block

Table 85. CPU Interface and Arbitration Logic Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface data and control signals			
cpu_addr[12:2]	11	In	CPU address bus. 11 bits are required to decode the address space for this block
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
diu_cpu_datain[31:0]	32	Out	Read data bus from the DIU to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_fc[2:0]	3	In	CPU Function Code signals.
cpu_diu_sel	1	In	Block select from the CPU. When *cpu_diu_sel* is high both *cpu_addr* and *cpu_dataout* are valid
diu_cpu_rdy	1	Out	Ready signal to the CPU. When *diu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *diu_cpu_datain* is valid.
diu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
DIU Read Interface to SoPEC Units			
<unit>_diu_rreq	1	In	SoPEC unit requests DRAM read.
DIU Write Interface to SoPEC Units			
<unit>_diu_wreq	1	In	SoPEC unit requests DRAM write.
Internal Inputs from other DAU blocks			
re_arbitrate	1	In	Signal telling the arbitration logic to choose the next arbitration winner.
debug_req	1	In	DIU request signal from Debug logic.
Internal Outputs from other DAU blocks			
debug_enable	17	Out	1 = Enable DIU Debug 0 = Disable DIU Debug
debug_start_adr[21:5]	17	Out	DIU debug start address.
debug_end_adr[21:5]	17	Out	DIU debug end address.
unit_gnt	1	Out	Signal lasting 1 cycle which indicates arbitration has occurred. Indicates *adr_sel* and *write_due* are valid.
adr_sel[4:0]	5	Out	Signal indicating which requesting SoPEC Unit has won arbitration.

Table 85. CPU Interface and Arbitration Logic Sub-block IO Definition

Port name	Pins	I/O	Description
write_due[1:0]	2	Out	*write_due[0]* indicates if the next arbitration winner will be a write access. *write_due[1]* indicates if the subsequent arbitration winner will be a write access. Valid on *unit_gnt*. *write_due[1]* is only required where 2 cycle random DRAM access is possible.
access_type[3:0]	4	Out	Signal indicating the origin of the winning arbitration 0000=main timeslot 0001=sub1timeslot 0010=sub2timeslot 0011=sub3timeslot 0100=sub4timeslot 0101=round-robin level1 0110=round-robin level2 0111=priority 1000=cpd round robin

20.13.5.1 CPU Interface and Arbitration Logic Sub-block Description

The CPU Interface and Arbitration Logic sub-block is shown in Figure 85. The CPU interface sub-block provides for the CPU to access DAU specific registers by reading or writing to the DAU address space.

The CPU subsystem bus interface is described in more detail in section Section 11.4.3. The DIU block will only allow supervisor mode read or write accesses to data space (i.e. *cpu_fc[2:0]* = b101). All other accesses will result in *diu_cpu_berr* being asserted.
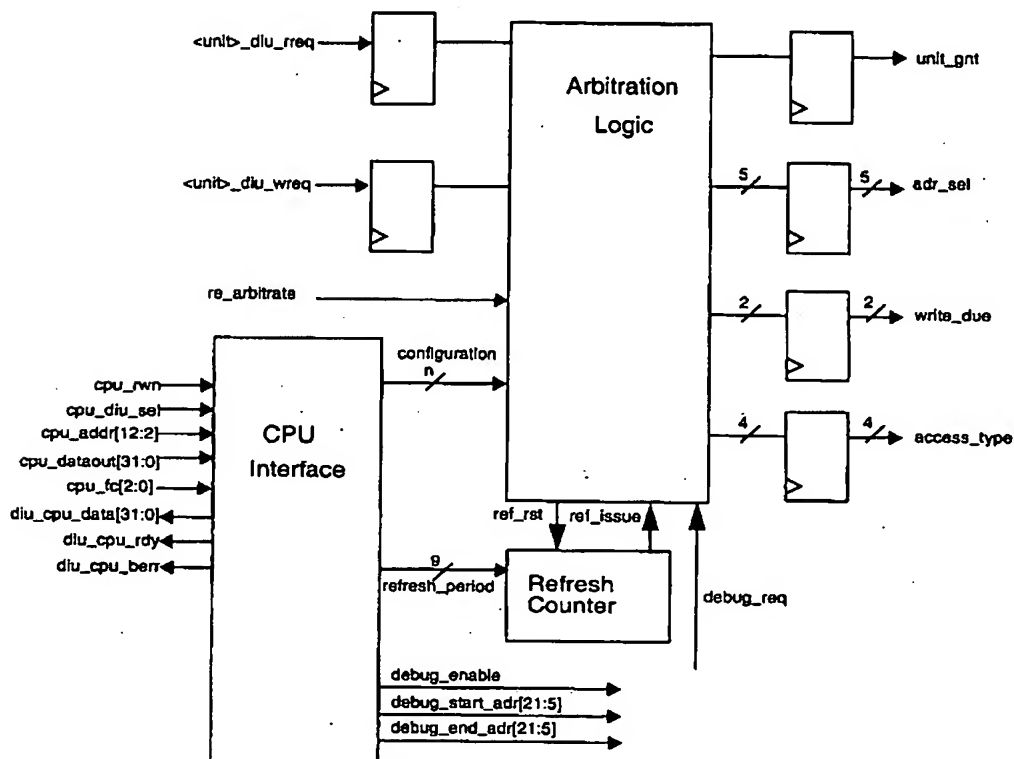
Figure 85. CPU Interface and Arbitration Logic

Arbitration is triggered by the signal *re_arbitrate* with the signal *unit_gnt* indicating that arbitration has occurred and the arbitration winner is indicated by *adr_sel[4:0]*. Arbitration should take 1 clock cycle so *unit_gnt* is asserted the clock cycle after *re_arbitrate* and stays high for 1 clock cycle. *adr_sel[4:0]* remains persistent until arbitration occurs again. The arbitration timing is shown in Figure 86.
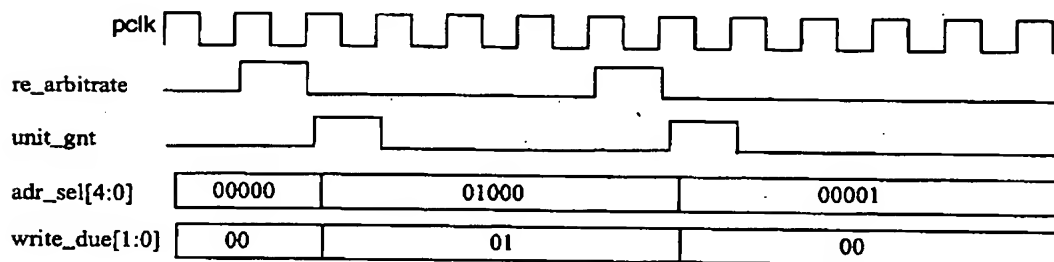

Figure 86. Arbitration timing

The basic arbitration table is 64 entries of 5 bits. Arbitration works by having a pointer advance to the next entry in the table whenever *re_arbitrate* is asserted. Four of the main slots can be assigned to a set of 4

SoPEC : Hardware Design

sub-slots. Each of these sub-slots requires 4 entries of 5 bits and a pointer. The pointer advances along the sub-entries every time arbitration selects a sub-slot. Slots can also be assigned to round-robin or priority allocations.

write_due[1:0] will point to timeslots 1 and 2 arbitration cycles in the future. *write_due[1:0]* looks ahead in the arbitration and indicates if the next arbitration winner (based on currently requesting write requesters) and the subsequent arbitration winner will be a write access. The *write_due[1:0]* functionality is added to allow write transactions be selected early by the command multiplexor sub-block to offset the latency of transferring the write data over the write data busses before the DRAM access can occur. This is only required for IBM DRAM. With Toshiba and Philips DRAM, the DRAM access latency masks any latency in transferring the write data.

If an assigned slot is not used (because its corresponding SoPEC Unit is not requesting) then it can be re-assigned in a number of ways. Slots can be re-assigned using a round-robin or a priority based assignment.

Round-robin assignment can have up to 21 round-robin slots. Its implementation requires a pointer to keep track of the currently assigned round-robin slot. If the slot pointed to is not requesting then the next round-robin slot is considered and so on. Round-robin is effectively a priority assignment with the slots assigned a priority according to the slot order. Round-robin and priority arbitration will be calculated in a hierarchical manner shown in Figure 87. If no round-robin slots are requesting then DRAM access is re-assigned according to priority.
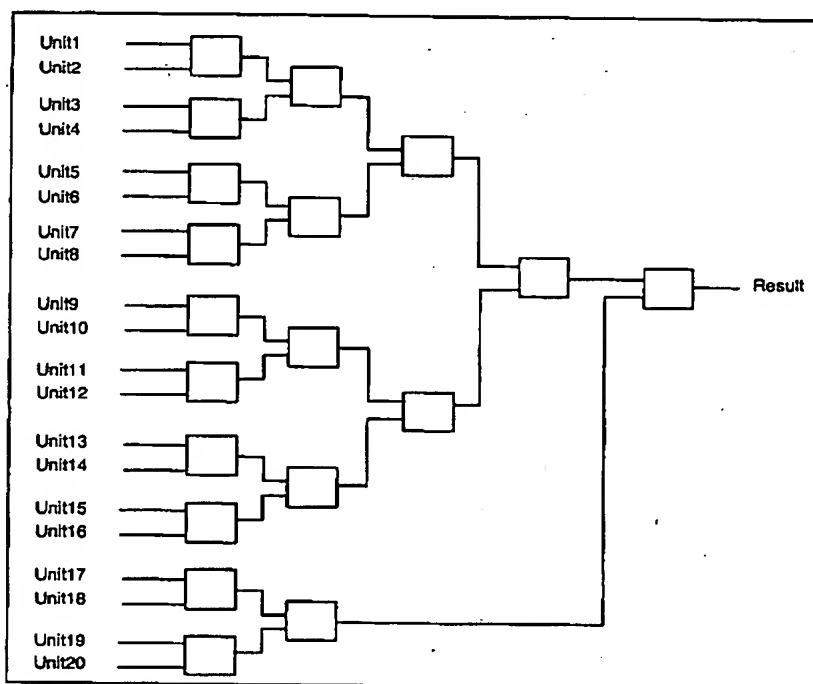

Figure 87. Hierarchical priority based arbitration calculation

It may be desirable to have 2 levels of round-robin arbitration. If there is no requester in the first level, then the arbitration looks at the second level. If there is no second-level requester then the DRAM access is assigned according to priority.

There is a background refresh counter which is reset whenever the arbitration logic selects a refresh. If the refresh counter reaches *refresh_period[9:0]* it stops counting and asserted a signal *ref_issue* which forces the next arbitration winner to be a refresh. This will delay any fixed timeslot allocation by one slot. If the slot was being re-allocated to a non-fixed timeslot requester then refresh will win the arbitration.

Debug can also request DIU access via the signal *debug_req*. DIU debug access when enabled will obtain DIU access at the expense of CPU DIU slots.

20.13.5.2 Arbitration of writes

For write accesses, except the CPU, 256-bits of write data are transferred from the SoPEC DIU write requestors over 64-bit write busses in 4 clock cycles. This write data transfer latency means that writes accesses must be arbitrated two timeslots in advance. Figure 88, which repeats the DIU write protocol shown in Figure 73, shows why this is necessary.

If this were a read access, then the read address is captured by the DIU in cycle 4 and presented to the DRAM in cycle 5. The read access at the DRAM will start in cycle 5. This corresponds to timeslot *n+2*. A write access cannot start until all the write data is available i.e. until cycle 9. This is a 4 cycle delay. The write access at the DRAM will not start until cycle 11 which corresponds to the start of timeslot *n+4*. Therefore, write arbitration must occur 2 timeslots in advance and incurs an additional latency of 2 cycles.

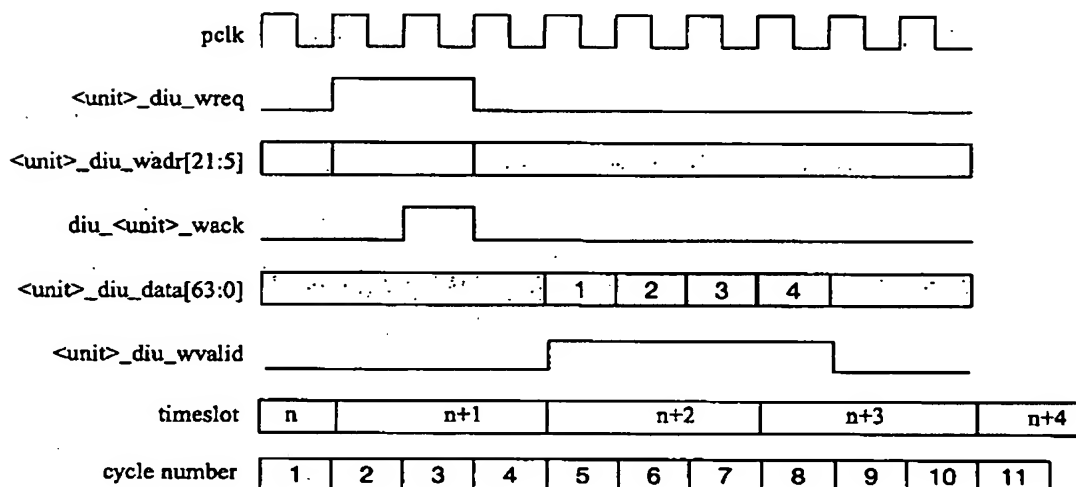The exact timing of read and write accesses will be outlined in Section 20.13 Implementation.

Figure 88. Write Protocol shown for a SoPEC Unit making a single 256-bit access

20.13.6 Command Multiplexor Sub-block

Table 86. Command Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DIU Read Interface to SoPEC Units			
<unit>_diu_radr[21:5]	17	In	Read address to DIU 17 bits wide (256-bit aligned word).
diu_<unit>_rack	1	Out	Acknowledge from DIU that read request has been accepted and new read address can be placed on <unit>_diu_radr
DIU Write Interface to SoPEC Units			
<unit>_diu_wadr[21:5]	17	In	Write address to DIU except CPU, SCB, CDU 17 bits wide (256-bit aligned word)
cpu_addr[21:0]	22	In	CPU Write address to DIU 22 bits wide (8-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
cdu_diu_wadr[21:3]	19	In	CDU Write address to DIU 19 bits wide (64-bit aligned word) Addresses cannot cross a 256-bit word DRAM boundary.
diu_<unit>_wack	1	Out	Acknowledge from DIU that write request has been accepted and new write address can be placed on <unit>_diu_wadr
debug_diu_wadr[21:5]	17	In	Debug write address to DIU 17 bits wide (256-bit aligned word)
diu_debug_wack	1	Out	Acknowledge from DIU that debug write request has been accepted and new write address can be presented.
Internal Inputs			
unit_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred.
adr_sel[4:0]	5	In	Signal indicating which requesting SoPEC Unit has won arbitration.
write_due[1:0]	2	In	*write_due[0]* indicates if the next arbitration winner will be a write access. *write_due[1]* indicates if the subsequent arbitration winner will be a write access. Valid on *unit_gnt*. *write_due[1]* is only required where 2 cycle random DRAM access is possible.
read_cmd_avail	1	In	Signal indicating that command multiplexor can issue read accesses.
write_cmd_avail	1	In	Signal indicating that command multiplexor can issue write accesses.
write_data_avail	1	In	Signal indicating that valid write data is available for the current command.
Internal Outputs			
re_arbitrate	1	Out	Signalling telling the arbitration logic to choose the next arbitration winner.
Signals from DCU			
dcu_cmd_accept	1	In	Signal indicating that the DCU has accepted a valid command from the DAU.

Table 86. Command Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
signals to DCU			
dau_cmd_avail	1	Out	Signal indicating a DAU command is available i.e. *dau_cmd_adr, dau_cmd_rwn* and *dau_cmd_refresh* are valid.
dau_cmd_adr[21:5]	17	Out	Signal indicating the address for the DRAM access. This is a 256-bit aligned DRAM address.
dau_cmd_rwn	1	Out	Signal indicating the direction for the DRAM access.
dau_cmd_refresh	1	Out	Signal indicating that a refresh command is to be issued. If asserted *cmd_adr* and *cmd_rwn* will be ignored.

20.13.6.1 DAU-DCU Interface Description

dau_cmd_avail indicates that the Command Multiplexor has a valid command to issue. When *dau_cmd_avail* is asserted the signals *dau_cmd_adr[21:5]*, *dau_cmd_rwn* and *dau_cmd_refresh* are valid. In the case of a write command, *dau_cmd_avail* will not be asserted until the Read-Write Data Multiplexor sub-block has valid write data to supply, indicated by *write_data_avail*, as well as a valid write address.

The DCU indicates that it has accepted a command by asserting *dcu_cmd_accept* for 1 cycle. This indicates to the Command Multiplexor that it can supply a new command to the DCU. The DCU cannot assert *dcu_cmd_accept* until the Command Multiplexor presents a valid command as indicated by *dau_cmd_avail*.

20.13.6.2 Command Multiplexor Sub-block Description

The command multiplexor sub-block issues read, write or refresh commands to the DCU, according to the SoPEC Unit selected for eDRAM access by the arbitration logic. The command multiplexor also signals the arbitration logic to perform arbitration to select the next SoPEC Unit for eDRAM access. Re-arbitration takes place, in general, when the DCU indicates on *dcu_cmd_accept* that it has accepted the previous command.

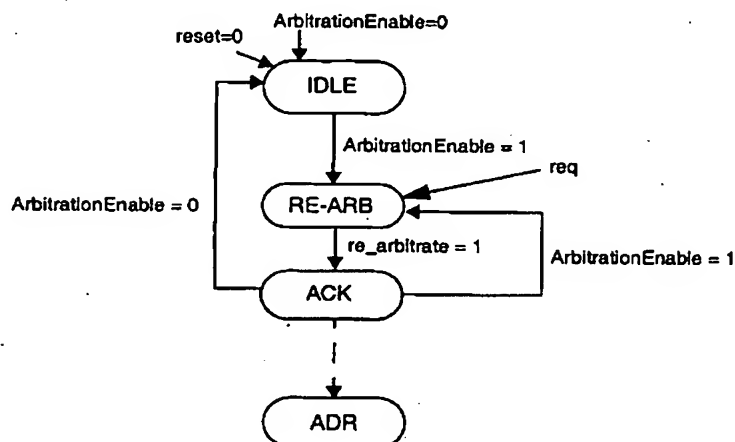A state-machine for the command multiplexor is shown in Figure 90.

Figure 90. Arbitration and Address Transfer state-machine

The states in Figure 90 are defined as follows:

Table 87. Command Multiplexor state description

State	Description
IDLE	Controller goes to this state on reset and when *ArbitrationEnable* is de-asserted.
RE-ARB	When *ArbitrationEnable* is asserted and there is a DIU requester, assert *re_arbitrate* so Arbitration Logic will select source of next DRAM access indicated by *adr_sel*.
ACK	Send acknowledge to source of next DRAM access indicated by *adr_sel*.
ADR	Receive DRAM address from source indicated by *adr_sel* and in the next cycle place it in command queue along with *adr_sel*.

In the *ACK* and *ADR* states of Figure 90, the signal *adr_sel* is used to multiplex between the SoPEC Units to capture the eDRAM address of the winning requester, as illustrated in Figure 91. The winning address is written into a command queue together with *adr_sel*. If the winning requester is *refresh* then no address is written into the command queue.
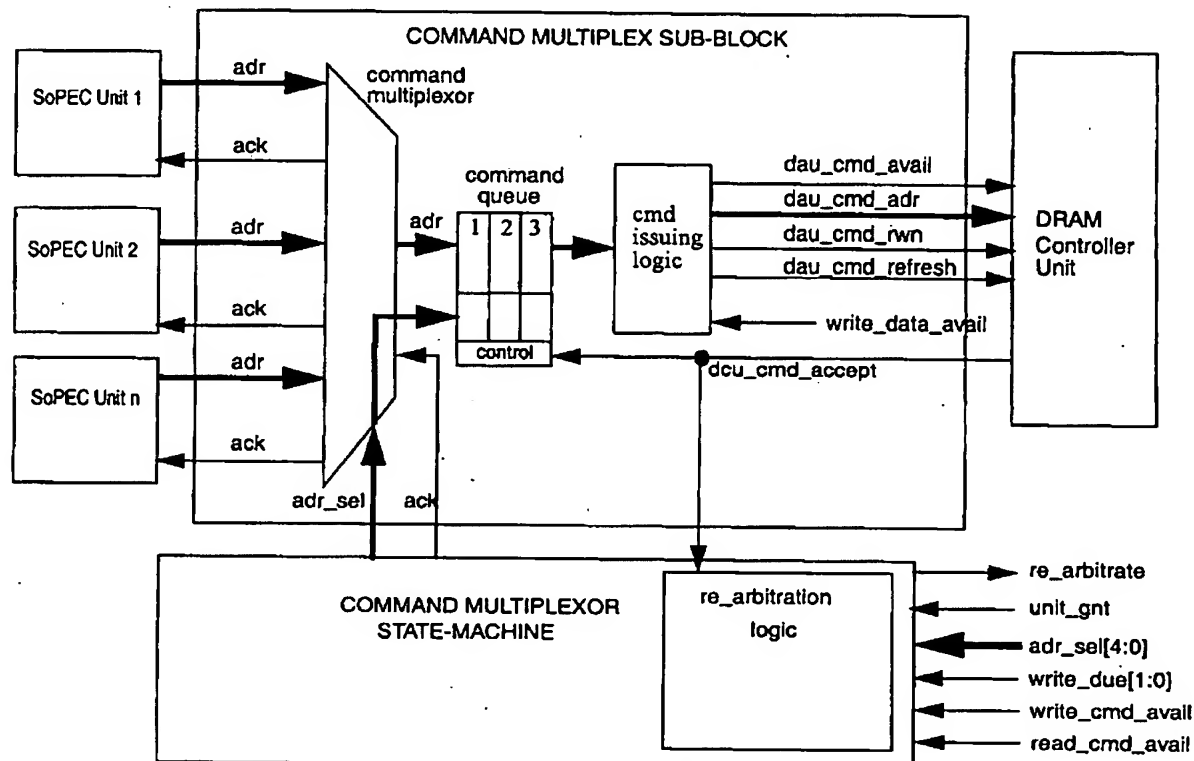
Figure 91. Command multiplexor sub-block

The command at the head of the command queue drives the command issuing logic which generates the signals required by the DRAM Controller Unit. The validity of the command is indicated by *dau_cmd_avail*. The commands are captured by the DCU by a *dcu_cmd_accept* strobe lasting 1 cycle. This signal also causes the command queue FIFO to be popped so that the next command is available to be be captured by the DCU. For a write command, *dau_cmd_avail* will not be asserted until there is also valid write data present. This is indicated by the signal *write_data_avail* from the Read-Write Data Multiplexor.

Normally, the command queue will only have one filled location i.e. *dcu_cmd_accept* will cause a command to be captured by the DCU and re-arbitration will be kicked off so as to provide the next command to the DCU in time for the next *dcu_cmd_accept* strobe. This is true for read and refresh commands. The timing is shown in Figure 92. It is assumed there is a pipeline delay between *dcu_cmd_accept* and *re_arbitrate* and a further pipeline delay between the address received from the SoPEC Unit and the address the DAU issues to the DCU.

For *refresh* commands:

```
re_arbitrate <= dcu_cmd_accept AND (command queue not full).
```

For read commands which use the shared read bus, we must also ensure that the read multiplexor logic is available to transmit the read data to the SoPEC read requester. This is indicated by the signal *read_cmd_avail* which provides flow control from the read data multiplexor logic. So for read commands:

```
re_arbitrate <= dcu_cmd_accept AND (command queue not full) AND
                (((cmd_adr_sel = shared read bus access) AND read_cmd_avail)
                OR (cmd_adr_sel = CPU))
```

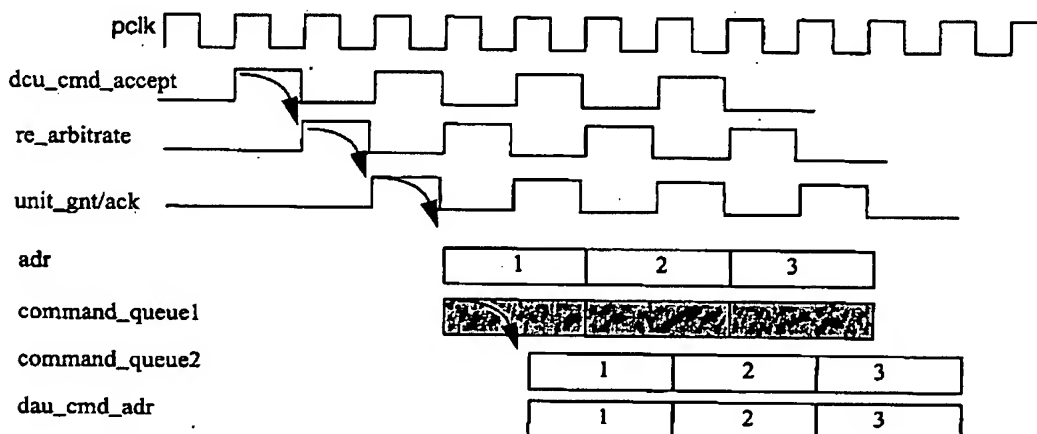A shared read bus access is any read access except the CPU.

Figure 92. Command Multiplexor sub-block timing for 2 cycle DRAM access read and refresh accesses

In the case of a write command, the write data must be transferred from the SoPEC requester before the write can occur. Arbitration should occur early to allow for any delay for the write data to be transferred. Figure 73 indicates that write data transfer over 64-bit busses will take a further 5 cycles after the address is transferred. The arbitration logic produces the signals *write_due[0]* and *write_due[1]* which point to timeslots 1 and 2 arbitration cycles in the future to indicate in advance future write accesses.

For Toshiba and Philips 8 or 9 cycle DRAM write access no such future write arbitration is required. In that case

```
re_arbitrate <= dcu_cmd_accept AND (command queue not full) AND (((cmd_adr_sel = shared
read bus access) AND read_avail) OR ((cmd_adr_sel = write access) AND write_cmd_avail)).
```

For 2 cycle random DRAM access and 5 cycle write data transfer latency both *write_due[0]* and *write_due[1]* are required. The command queue must be sized at 3 deep. If the timeslot sequence for DRAM access is a read, followed by a write, followed by a write, then arbitration should occur to select the read access immediately followed by re-arbitration twice more to select the future write accesses. The condition for re-arbitration in advance of the write access is

```
re_arbitrate <= (command queue not full) AND write_cmd_avail AND (write_due[0] OR
write_due[1]).
```

write_cmd_avail provides flow control from the write data multiplexor logic.

Open Issue

The mechanism described here only pre-empts write accesses indicated by *write_due[1:0]* based on fixed timeslot allocation. If a write access is selected based on un-used timeslot re-allocation then it will introduce a latency in the access stream. One possibility to only disallow un-used timeslot allocation in the case of write accesses. An alternative is to use 256-bit data busses to transfer write data but this is likely to cause an increase the area of the DIU and/or decrease the chip wiring utilization. Another possibility is to have separate un-used timeslot reallocation logic for write accesses associated with *write_due[1:0]* i.e. write requests are always only considered 1 or 2 arbitration cycles in the future. In this case there is effectively a different time window for considering write requests and read requests.

20.13.7 Read and Write Multiplexor Sub-block

Table 88. Read and Write Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DIU Read Interface to SoPEC Units			
diu_data[63:0]	64	Out	Data from DIU to SoPEC Units except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
diu_cpu_data[255:0]	256	Out	256-bit data from DIU to CPU.
diu_<unit>_rvalid	1	Out	Signal from DIU telling SoPEC Unit that valid read data is on the diu_data bus
DIU Write Interface to SoPEC Units			
<unit>_diu_data[63:0]	64	In	Data from SoPEC Unit to DIU except CPU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
cpu_diu_data[31:0]	32	In	Data from CPU to DIU.
cpu_addr[4:0]	5	In	Lower bits of CPU Write address to indicate which byte within the 256-bit DRAM word is selected.
cpu_diu_wmask[1:0]	2	In	Flag indicating format of CPU write to DRAM cpu_diu_wmask = "00": 8-bit write cpu_diu_wmask = "01": 16-bit write cpu_diu_wmask = "10": 32-bit write cpu_diu_wmask = "11": reserved cpu_addr[2:0] are driven in accordance with the width of the data access indicated by cpu_diu_wmask. Addresses cannot cross a 256-bit word DRAM boundary.
<unit>_diu_wvalid	1	In	Signal from SoPEC Unit indicating that data on <unit>_diu_data is valid.
Internal Inputs			
unit_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred.
adr_sel[4:0]	5	In	Signal indicating which requesting SoPEC Unit has won arbitration.
Internal Outputs			
read_cmd_avail	1	Out	Signal indicating that command multiplexor can issue read accesses.
write_cmd_avail	1	Out	Signal indicating that command multiplexor can issue write accesses.
write_data_avail	1	Out	Signal indicating that valid write data is available for the current command.
DCU Inputs			
dcu_rdata	256	In	256-bit read data from DCU.
dcu_rrvalid	1	In	Signal indicating valid read data on dcu_rdata.
DCU Outputs			

Table 88. Read and Write Multiplexor Sub-block IO Definition

Port name	Pins	I/O	Description
dau_wdata	256	Out	256-bit write data to DCU
dau_wmask	256	Out	256-bit write data mask to DCU for IBM DRAM (byte masks are used for Phillips and Toshiba DRAM).
dau_wvalid	17	Out	Signal indicating valid write data and write mask.
Debug signals			
debug_wdata	256	In	256-bit debug write data
debug_wvalid	1	In	256-bit debug write data valid
read_adr_sel[4:0]	5	Out	Signal indicating the SoPEC Unit for which the current read transaction is occurring.
read_complete	1	Out	Signal indicating that read transaction to SoPEC Unit indicated by *read_adr_sel* is complete.
write_adr_sel[4:0]	5	Out	Signal indicating the SoPEC Unit for which the current write transaction is occurring.
write_complete	1	Out	Signal indicating that write transaction to SoPEC Unit indicated by *write_adr_sel* is complete.

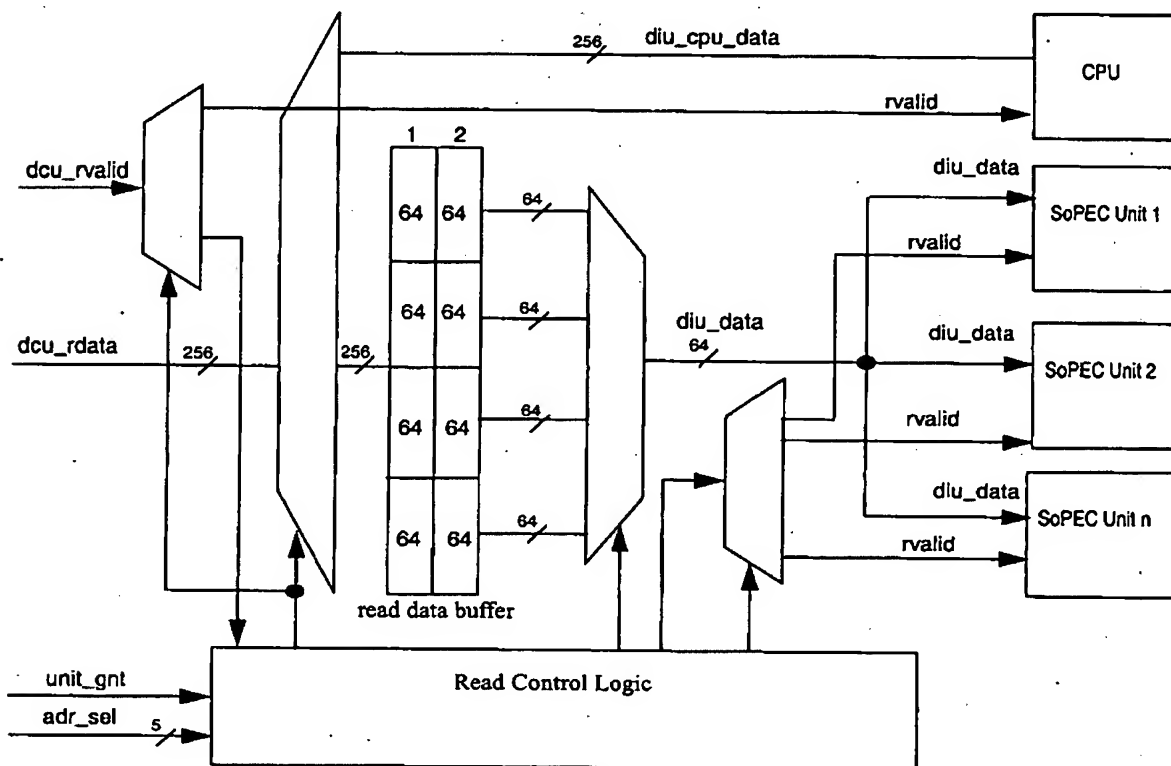## 20.13.7.1 Read Multiplexor logic description

Figure 93. Read multiplexor logic

There are 2 read channels - one for the CPU and a shared read bus for the rest of SoPEC. The shared read bus has buffering for 2 times 256-bits of read data i.e 256-bits of data can be received from the DCU while data is being transferred over the 64-bit shared read bus to the SoPEC Units. Once a read address is issued by the arbitration logic the *adr_sel[4:0]* value is put into a read command queue in the read control logic. The queued *adr_sel[4:0]* values allow the *dcu_rvalid* and read data from the DCU to be directed to the correct source. In the case of the CPU bus *dcu_rvalid* and *dcu_data* can be multiplexed by the *adr_sel[4:0]* value at the head of the FIFO directly to the CPU. If the incoming data goes over the shared read channel then the data is stored in a 2 deep 256-bit read data buffer and output over 4 cycles to the SoPEC requester when the previous transaction on the shared read bus is complete.

The depth of the *adr_sel[4:0]* read command queue is 2. When the queue is full no further *adr_sel[4:0]* can be accepted and no further read commands can be issued by the command multiplexor to the DCU. This provides flow control back to the re-arbitration logic in the command multiplexor. The signal *read_cmd_avail* indicates that spaces are available in the read command queue

The 2 deep command queue and the double data buffer means that the access rate will be limited to which ever takes longer - DRAM access or transfer of read data over the shared read data bus. Some extra logic may need to be added to time the assertion of *read_cmd_avail* so that the read latency is kept to a minimum.

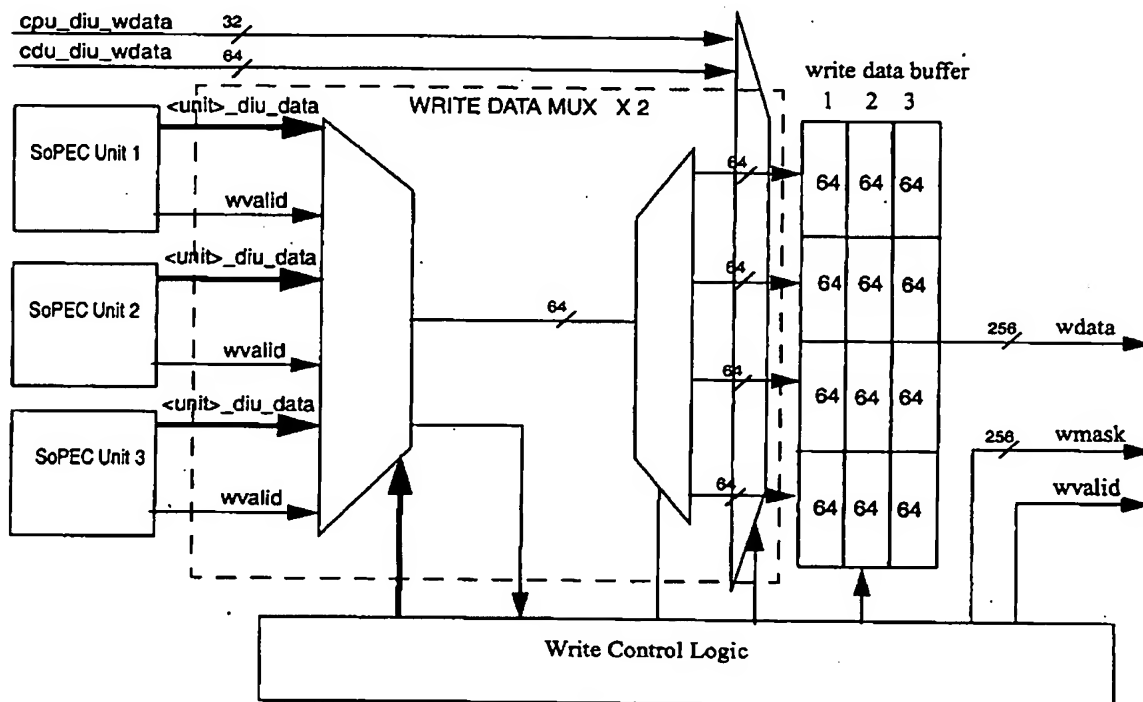### 20.13.7.2 Write Multiplexor logic description

Figure 94.

Once a write address is issued by the arbitration logic, the *adr_sel[4:0]* value is put into a write command queue FIFO in the write control logic. The queued *adr_sel[4:0]* values allow the *wvalid* and write data from the SoPEC requester to be multiplexed to the DCU. The write multiplex logic is duplicated 2 times to provide two overlapping write channels. If 256-bit write data busses are used then a single write channel which can be shared by CPU is all that is required.

The depth of the *adr_sel[4:0]* queue is 3. When the queue is full no further *adr_sel[4:0]* values from the CPU Arbitration block will be accepted. This provides flow control back to the re-arbitration logic in the Command Multiplexor sub-block. The 2 channels cannot select the same SoPEC write requester. *write_cmd_avail* is asserted whenever there is a space in the queue. There are 2 write pointers and 1 read pointer. Each of the channels has a write pointer associated with it. *write_data_avail* indicates that valid write data is available to be issued along with the address the command multiplexor will issue.

There are 2 special cases for write accesses - CPU writes and CDU writes.

CPU writes

In the case of CPU writes the CPU write data bus is only 32-bits wide. *cpu_diu_wmask[1:0]* indicates how. many bits have to be written: 8, 16 or 32-bits. The associated address *cpu_addr[21:0]* is a byte aligned address. The actual DRAM write must be a 256-bit access. The command multiplexor issues the 256-bit DRAM address *cpu_addr[21:5]*. *cpu_addr[4:0]* and *cpu_diu_wmask[1:0]* are used to calculate the bit write mask *wmask[255:0]* for the write access.

```
8-bit write: cpu_diu_wmask[1:0] = '00':
     bits 8*cpu_addr[4:0] to (8*(cpu_addr[4:0]+1) -1) of wmask[255:0] are asserted.
16-bit write: cpu_diu_wmask[1:0] = '01':
     bits 8*cpu_addr[4:0] to (16*(cpu_addr[4:0]+1) -1) of wmask[255:0] are asserted.
32-bit write: cpu_diu_wmask[1:0] =. '10':
     bits 8*cpu_addr[4:0] to (32*(cpu_addr[4:0]+1) -1) of wmask[255:0] are asserted.
```

CDU writes

Each CDU write access is a burst of 4 times 64-bits of write data to the 64-bits of the 256-bit DRAM address indicated by *cdu_diu_wadr[21:3]* and the 3 subsequent 256-bit DRAM words. If these 4 DRAM words lie in the same DRAM row then an efficient access will be obtained. The command multiplexor logic must issue 4 successive accesses to 256-bit DRAM addresses *cdu_diu_wadr[21:5], +1, +2, +3.* *wmask[255:0]* is calculated using *cdu_diu_wadr[4:3]* i.e. bits $64*cdu\_diu\_wadr[4:3]$ to $(64*(cdu\_diu\_wadr[4:3]+1) -1)$ are asserted.

20.13.8 DIU Debug Sub-Block

Table 9. DIU ug Sub-block IO Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
CPU Interface and Arbitration Logic			
debug_enable	1	In	1 = Enable DIU Debug 0 = Disable DIU Debug
debug_start_adr[21:5]	17	In	DIU debug start address.
debug_end_adr[21:5]	17	In	DIU debug end address.
debug_req	1	Out	DIU request signal from Debug logic.
<unit>_diu_rreq	1	In	SoPEC unit requests DRAM read. A read request must be accompanied by a valid read address.
<unit>_diu_wreq	1	In	SoPEC unit requests DRAM write. A write request must be accompanied by a valid write address.
unit_gnt	1	In	Signal lasting 1 cycle which indicates arbitration has occurred. Indicates *adr_sel* and *write_due* are valid.
adr_sel[4:0]	5	In	Signal indicating which requesting SoPEC Unit has won arbitration.
access_type[3:0]	4	In	Signal indicating the origin of the winning arbitration 0000=maintimeslot 0001=sub1timeslot 0010=sub2timeslot 0011=sub3timeslot 0100=sub4timeslot 0101=round-robin level1 0110=round-robin level2 0111=priority 1000=cpd round robin
DRAM Control Unit			
dcu_refresh_complete	1	In	Signal indicating that the DCU has completed a refresh. Exact timing needs to be defined.
Read Write Data Multiplexor			
debug_wdata	256	Out	256-bit debug write data
debug_wvalid	1	Out	256-bit debug write data valid
read_adr_sel[4:0]	5	In	Signal indicating the SoPEC Unit for which the current read transaction is occurring.
read_complete	1	In	Signal indicating that read transaction to SoPEC Unit indicated by *read_adr_sel* is complete.
write_adr_sel[4:0]	5	In	Signal indicating the SoPEC Unit for which the current write transaction is occurring.
write_complete	1	In	Signal indicating that write transaction to SoPEC Unit indicated by *write_adr_sel* is complete.

External visibility of the DIU must be provided for debug purposes. To allow this special debug logic is added to the DIU. When DIU debug is enabled by *debug_enable*, the DIU debug sub-block will collect debug data. The DIU debug sub-block will itself have a 256-bit double buffer interface to the DIU. When

256-bits of debug data have been collected then the DIU debug sub-block will request access to DRAM by asserting *debug_req*. When the request is acknowledged on *diu_debug_wack* the DIU debug circuit will supply a write address, 256-bits of write data and a write valid. The arbitration logic will give the DIU Debug priority over CPU slots. This means the Debug 256-bit double-buffer will never overflow and debug will not affect the access of other blocks except the CPU (but this should not be important as debug will request with a low frequency and there are many timeslots assigned to the CPU). The DIU Debug circuit will generate a write address by incrementing (and wrapping around) between *debug_start_adr[21:5]* and *debug_end_adr[21:5]*.

Two kinds of debug information seem sensible to gather.

a. The order and source of DIU requesters winning. This is obtained by storing *adr_sel[4:0]* along with the *access_type[3:0]* every time *unit_gnt* is asserted.

b. The time between a DIU requester requesting an access and completing the access. This information is obtained by having a counter for each DIU requester. The counter is reset and starts counting when the Unit starts requesting. The count is reset when the read or write access is complete as indicated by *read_complete* AND *read_adr_sel[4:0]* OR *write_complete* AND *write_adr_sel[4:0]*. When *refresh* is complete this is indicated by *dcu_refresh_complete*. Typically most SoPEC DIU requesters require an access every 256 cycles so a 10-bit counter is sufficient for most requesters. HCU and TE(TFS) require only a few accesses per line so in this case 15-bit counters are adequate. The count is returned along with the index of the SoPEC Unit.

The two kinds of debug information need to be both written to the DRAM debug channel. This can be achieved by filling separate 256-bit double buffers with the 2 sources of information. Each 256-bit word may contain un-used bits depending on the packing. The first bit of each 256-bit debug word will indicate which of the two kinds of debug information is contained therein (0 for *a*, 1 for *b* above).

S3

PEP SUBSYSTEM

21 PEP Controller Unit (PCU)

21.1 OVERVIEW

The PCU has three functions:
- The first is to act as a bus bridge between the CPU-bus and the PCU-bus for reading and writing PEP configuration registers.
- The second is to support page banding by allowing the PEP blocks to be reprogrammed between bands by retrieving commands from DRAM instead of being programmed directly by the CPU.
- The third is to send register debug information to the RDU, within the CPU subsystem, when the PCU is in Debug Mode.

21.2 INTERFACES BETWEEN PCU AND OTHER UNITS
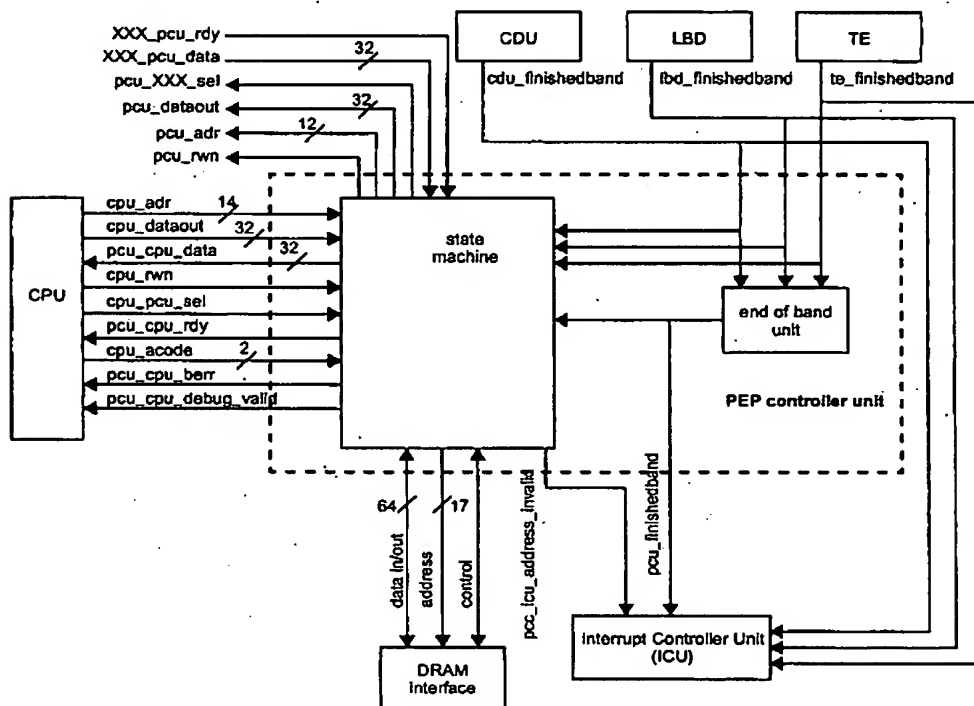
Figure 95. Block diagram of PCU

21.3 BUS BRIDGE

The PCU is a bus-bridge between the CPU-bus and the PCU-bus. The PCU is a slave on the CPU-bus but is the only master on the PCU-bus. See Figure 13 on page 39.

21.3.1 CPU accessing PEP

All the blocks in the PEP can be addressed by the CPU via the PCU. The MMU in the CPU-subsystem will decode a PCU select signal, *cpu_pcu_sel*, for all the PCU mapped addresses (see section 11.4.3 on page 70). Using *cpu_adr* bits 15-12 the PCU will decode individual block selects for each of the blocks within the PEP. The PEP blocks then decode the remaining address bits needed to address their PCU-bus mapped registers. Note: the CPU is only permitted to perform supervisor-mode data-type accesses of the PEP, i.e. *cpu_acode* = 11. If the PCU is selected by the CPU and any other code is present on the *cpu_acode* bus the access is ignored by the PCU and the *pcu_cpu_berr* signal is strobed,

CPU commands have priority over DRAM commands. When the PCU is executing each set of four commands retrieved from DRAM the CPU can access PCU-bus registers. In the case that DRAM commands are being executed and the CPU resets the *CmdSource* to zero, the contents of the DRAM *CmdFifo* is invalidated and no further commands from the fifo are executed. The *CmdPending* and *NextBandCmdEnable* work registers are also cleared.

21.4 PAGE BANDING

The PCU can be programmed to associate microcode in DRAM with each *finishedband* signal. When a *finishedband* signal is asserted the PCU will read commands from DRAM and execute these commands. These commands are each 64-bits (see Section 21.8.5) and consist of 32-bit address bits and 32 data bits and allow PCU mapped registers to be programmed directly by the PCU.

If more than one *finishedband* signal is received at the same time, or others are received while microcode is already executing, the PCU will hold the commands as pending, and will execute them at the first opportunity.

Each microcode program associated with *cdu_finishedband*, *lbd_finishedband* and *te_finishedband* would simply restart the appropriate unit with new addresses - a total of about 4 or 5 microcode instructions. As well, or alternatively, *pcu_finishedband* can be used to set up all of the units and therefore involves many more instructions. This minimizes the time that a unit is idle in between bands. The *pcu_finishedband* control signal is issued once the specified combination of CDU, LBD and TE (programmed in *BandSelectMask*) have finished their processing for a band.

21.5 INTERRUPTS, ADDRESS LEGALITY AND SECURITY

Interrupts are generated when the various page expansion units have finished a particular band of data from DRAM. The *cdu_finishedband*, *lbd_finishedband* and *te_finishedband* signals are combined in the PCU into a single interrupt *pcu_finishedband* which is exported by the PCU to the interrupt controller.

The PCU mapped registers should only be accessible from Supervisor Data Mode. The area of DRAM where PCU commands are stored should be a Supervisor Mode only DRAM area. Configuration register address legality is not enforced by the MMU i.e. the MMU does not check if the block address points to a valid PEP subsystem block. When the PCU is executing commands from CPU, any block-address decoded from a command which is not part of the PEP block-address map will cause the PCU to ignore the command and strobe the *pcu_invalid_address* interrupt signal. The CPU can then interrogate the PCU to find the source of the illegal command.

When the PCU is executing commands from DRAM, any address decoded from a command which is not part of the PEP address map will cause the PCU to:
- Cease execution of current command and flush all remaining commands already retrieved from DRAM.
- Clear *CmdPending* work-register.
- Clear *NextBandCmdEnable* registers.

- Set *CmdSource* to zero.

In addition to cancelling all current and pending DRAM accesses the PCU strobes the *pcu_invalid_address* interrupt signal. The CPU can then interrogate the PCU to find the source of the illegal command.

21.6 DEBUG MODE

When the need to monitor the (possibly changing) value in any PEP configuration register the PCU may be placed in Debug Mode. This is done via the CPU setting certain Debug Address and Debug Enable registers within the PCU. Once in Debug Mode the PCU continually performs read accesses of the target PEP configuration register (following the protocol detailed in Section 21.8.2) and sends the read value to the RDU. Debug Mode has the lowest priority of all PCU functions: if the CPU wishes to perform an access or there are DRAM commands to be executed they will interrupt the Debug access, and the PCU will resume Debug access once a CPU or DRAM command has completed.

21.7 IMPLEMENTATION

21.7.1 Definitions of I/O

Table 90. PCU Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC functional clock
prst_n	1	In	Active-low, synchronous reset in pclk domain
End of Band Functionality			
cdu_finishedband	1	In	Finished band signal from CDU
lbd_finishedband	1	In	Finished band signal from LBD
te_finishedband	1	In	Finished band signal from TE
pcu_finishedband	1	Out	Asserted once the specified combination of CDU, LBD, and TE have finished their processing for a band.
PCU address error			
pcu_icu_address_invalid	1	Out	Strobed if PCU decodes a non PEP address from commands retrieved from DRAM or CPU.
CPU Subsystem Interface Signals			
cpu_adr[15:2]	14	In	CPU address bus. 14 bits are required to decode the address space for the PEP.
cpu_dataout[31:0]	32	In	Shared write data bus from the CPU
pcu_cpu_data[31:0]	32	Out	Read data bus to the CPU
cpu_rwn	1	In	Common read/not-write signal from the CPU
cpu_acode[1:0]	2	In	CPU Access Code signals. These decode as follows: 00 - User program access 01 - User data access 10 - Supervisor program access 11 - Supervisor data access
cpu_pcu_sel	1	In	Block select from the CPU. When *cpu_pcu_sel* is high both *cpu_adr* and *cpu_dataout* are valid
pcu_cpu_rdy	1	Out	Ready signal to the CPU. When *pcu_cpu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *cpu_dataout* has been registered by the block and for a read cycle this means the data on *pcu_cpu_data* is valid.
pcu_cpu_berr	1	Out	Bus error signal to the CPU indicating an invalid access.
pcu_cpu_debug_valid	1	Out	Debug Data valid on *pcu_cpu_data* bus. Active high.
PCU Interface to PEP blocks			
pcu_adr[11:2]	10	Out	PCU address bus. The 10 least significant bits of *cpu_adr [15:2]* allow 1024 32-bit word addressable locations per PEP block. Only the number of bits required to decode the address space are exported to each block.
pcu_dataout[31:0]	32	Out	Shared write data bus from the PCU
<unit>_pcu_datain[31:0]	32	In	Read data bus from each PEP subblock to the PCU
pcu_rwn	1	Out	Common read/not-write signal from the PCU

Table 90. PCU Port List

Port Name	Pins	I/O	Description
pcu_<unit>_sel	1	Out	Block select for each PEP block from the PCU. Decoded from the 4 most significant bits of *cpu_adr[15:2]*. When *pcu_<unit>_sel* is high both *pcu_adr* and *pcu_dataout* are valid
<unit>_pcu_rdy	1	In	Ready from each PEP block signal to the PCU. When *<unit>_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *<unit>_pcu_datain* is valid.
DIU Read Interface signals			
pcu_diu_rreq	1	Out	PCU requests DRAM read. A read request must be accompanied by a valid read address.
pcu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_pcu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *pcu_diu_radr*
diu_data[63:0]	64	In	Data from DIU to PCU. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
diu_pcu_rvalid	1	In	Signal from DIU telling PCU that valid read data is on the *diu_data* bus

21.7.2 Configuration Registers

Table 91. PCU Configuration Registers

Address PCU base +	register	# bits	reset	description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the PCU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	CmdAdr[21:5] (256-bit aligned DRAM address)	17	0x00 000	The address of the next set of commands to retrieve from DRAM. When this register is written to, either by the CPU or DRAM command, 1 is also written to *Cmd-Source* to cause the execution of the commands at the specified address.
0x08	BandSelectMask[2:0]	3	0x0	Selects which input finishedBand flags are to be watched to generate the combined finishedAll-Band signal. Bit0 - lbd_finishedband Bit1 - cdu_finishedband Bit2 - te_finishedband
0x0C, 0x10, 0x14, 0x18	NextBandCmdAdr[3:0][21:5] (256-bit aligned DRAM address)	4x17	0x00 000	The address to transfer to *CmdAdr* as soon as possible after the next *finishedBand[n]* signal has been received as long as *NextBandCmdEnable[n]* is set. A write from the PCU to *NextBandCmdAdr[n]* with a non-zero value also sets *NextBandCmdEnable[n]*. A write from the PCU to *NextBandCmdAdr[n]* with a 0 value clears *NextBandCmdEnable[n]*.
0x20	CmdSource	1	0x0	0 - commands are taken from the CPU 1 - commands are taken from the CPU as well as DRAM at CmdAdr.
0x24	DebugSelect[15:2]	14	0x00 00	Debug address select. Indicates the address of the register to report on the *pcu_cpu_data* bus when it is not otherwise being used, and the PEP bus is not being used Bits [15:12] select the unit (see Table 92) Bits [11:2] select the register within the unit
Work registers (read only)				
0x28	InvalidAddress[21:3] (64-bit aligned DRAM address)	19	0	Address of illegal 64-bit command in DRAM. Only valid when *pcu_icu_address_invalid* has been strobed. (64-bit aligned address)
0x2C	CmdPending	4	0	For each bit: 0 - no commands pending for NextBandCmd[n] 1 - commands pending for NextBandCmdAdr[n] Read only register.

Table 91. PCU Configuration Registers

Address PCU base+	register	#bits	reset	description
0x34	FinishedSoFar	3	0x0	The appropriate bit is set whenever the corresponding input finishedBand flag is set and the corresponding bit in the BandSelectMask bit is also set. If all FinishedSoFar bits are set wherever Band-Select bits are also set, all FinishedSoFar bits are cleared and the output finishedAllBand signal is given. **Read only register.**
0x8	NextBandCmdEnable	4	0x0	This register can be written to indirectly (i.e. the bits are set or cleared via writes to *NextBandCmdAdr[n]*) For each bit: 0 - do nothing at the next *finishedBand[n]* signal. 1 - Execute instructions at *NextBandCmdAdr[n]* as soon as possible after receipt of the next *finishedBand[n]* signal. Bit0 - lbd_finishedband Bit1 - cdu_finishedband Bit2 - te_finishedband Bit 3 - finishedAllBand **Read only register.**

21.8   DETAILED DESCRIPTION

21.8.1   PEP Blocks Register Map

All PEP accesses are 32-bit register accesses.

From Table 92 it can be seen that four bits only are necessary to address each of the sub-blocks within the PEP part of SoPEC. Up to 14 bits may be used to address any configurable 32-bit register within PEP. This gives scope for 1024 configurable registers per sub-block. This address will come either from the CPU or from a command stored in DRAM. The bus is assembled as follows:

- adr[15:12] = sub-block address
- adr[n:2] = 32-bit register address within sub-block, only the number of bits required to decode the registers within each sub-block are used.

Table 92. PEP blocks Register Map

Block	Block Select Decode Cpu_adr[15:12]
PCU	0x0
CDU	0x1
CFU	0x2
LBD	0x3
SFU	0x4
TE	0x5
TFU	0x6
HCU	0x7
DNC	0x8

Table 92. PEP blocks Register Map

Block	Block Select Decode cpu_adr[15:12]
DWU	0x9
LLU	0xA
PHI	0xB
Reserved	0xC to 0xF

21.8.2   Internal PCU PEP protocol

The PCU performs PEP configuration register accesses via a select signal, *pcu_<block>_sel*. The read/write sense of the access is communicated via the *pcu_rwn* signal (1 = read, 0 = write). Write data is clocked out, and read data clocked in upon receipt of the appropriate select-read/write-address combination.
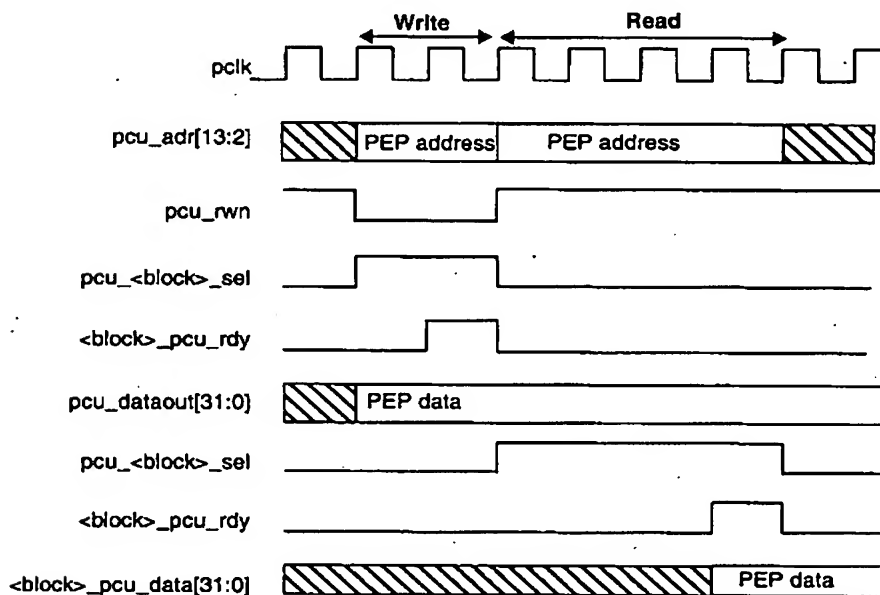
Figure 96. PCU accesses to PEP registers

Figure 96 shows a write operation followed by a read operation. The read operation is shown with wait states while the PEP block returns the read data.

For access to the PEP blocks a simple bus protocol is used. The PCU first determines which particular PEP block is being addressed so that the appropriate block select signal can be generated. During a write access PCU write data is driven out with the address and block select signals in the first cycle of an access. The addressed PEP block responds by asserting its ready signal indicating that it has registered the write data and the access can complete. The write data bus is common to all PEP blocks.

A read access is initiated by driving the address and select signals during the first cycle of an access. The addressed PEP block responds by placing the read data on its bus and asserting its ready signal to indicate

to the PCU that the read data is valid. Each block has a separate point-to-point data bus for read accesses to avoid the need for a tri-stateable bus.

21.8.3   PCU DRAM access requirements

The PCU can execute register programming commands stored in DRAM. These commands can be executed at the start of a print run to initialize all the registers of PEP. The PCU can also execute instructions at the start of a page, and between bands. In the inter-band time, it is critical to have the PCU operate as fast as possible. Therefore in the inter-page and inter-band time the PCU needs to get low latency access to DRAM.

A typical band change requires on the order of 4 commands to restart each of the CDU, LBD, and TE, followed by a single command to terminate the DRAM command stream. This is on the order of 5 commands per restart component.

The PCU does single 256 bit reads from DRAM. Each PCU command is 64 bits so each 256 bit DRAM read can contain 4 PCU commands. The requested command is read from DRAM together with the next 3 contiguous 64-bits which are cached to avoid unnecessary DRAM reads. Writing zero to *CmdSource* causes the PCU to flush commands and terminate program access from DRAM for that command stream. The PCU requires a 256-bit buffer to the 4 PCU commands read by each 256-bit DRAM access. When the buffer is empty the PCU can request DRAM access again. Adding a 256-bit double buffer would allow the next set of 4 commands to be fetched from DRAM while the current commands are being executed.

1024 commands of 64 bits requires 8 kB of DRAM storage.

Programs stored in DRAM are referred to as *PCU Program Code*.

21.8.4   End of band unit

The state machine is responsible for watching the various input *xx_finishedband* signals, setting the *FinishedSoFar* flags, and outputting the *finished_all_band* flags as specified by the *BandSelect* register.

Each cycle, the end of band unit performs the following tasks:

```
finishedAllBand = (FinishedSoFar[0] == BandSelectMask[0]) AND
                  (FinishedSoFar[1] == BandSelectMask[1]) AND
                  (FinishedSoFar[2] == BandSelectMask[2]) AND
                  (BandSelectMask[0] OR BandSelectMask[1] OR BandSelectMask[2])
  if (finishedAllBand == 1) then
    FinishedSoFar[0] = 0
.   FinishedSoFar[1] = 0
    FinishedSoFar[2] = 0
  else
    FinishedSoFar[0] = FinishedSoFar[0] OR (lbd_finishedband AND BandSelectMask[0])
    FinishedSoFar[1] = FinishedSoFar[1] OR (cdu_finishedband AND BandSelectMask[1])
    FinishedSoFar[2] = FinishedSoFar[2] OR (te_finishedband AND BandSelectMask[2])
```

Note that it is the responsibility of the microcode at the start of printing a page to ensure that all 3 *FinishedSoFar* bits are cleared. It is not necessary to clear them between bands since this happens automatically.

21.8.5 Executing commands from DRAM

Registers in PEP can be programmed by means of simple 64-bit commands fetched from DRAM. The format of the commands is given in Table 93. Register locations can have a data value of up to 32 bits. Commands are PEP register write commands only.

Table 93. Register write commands in PEP

command	bits 63-32	bits 31-16	bits 15-2	bits 1-0
Register write	data	zero	32-bit word address	zero

Due attention must be paid to the endianness of the processor. The LEON processor is a big-endian processor (bit 7 is the most significant bit).

21.8.6 General Operation

Upon a Reset condition, *CmdSource* is cleared (to 0), which means that all commands are initially sourced only from the CPU bus interface. Registers and can then be written to or read from one location at a time via the CPU bus interface.

If *CmdSource* is 1, commands are sourced from the DRAM at *CmdAdr* or from the CPU bus. Writing an address to *CmdAdr* automatically sets *CmdSource* to 1, and causes a command stream to be retrieved from DRAM. The PCU will execute commands from the CPU or from the DRAM command stream, giving higher priority to the CPU always.

Regardless of the state of *CmdSource* the DRAM requestor is examines the *CmdPending* bits to determine if a new DRAM command stream is pending. If any of *CmdPending* bits are set, then the appropriate *NextBandCmdAdr* is copied to *CmdAdr* (causing *CmdSource* to get set to 1) and a new command DRAM stream is retrieved from DRAM and executed by the PCU. Note that a new DRAM command stream only gets retrieved when the current command stream is empty.

If there are no DRAM commands pending, and no CPU commands the PCU defaults to an idle state. When idle the PCU address bus defaults to the *DebugSelect* register value (bits 11 to 2 in particular) and the default unit PCU data bus is reflected to the CPU data bus. The default unit is determined by the *DebugSelect* register bits 15 to 12.

In conjunction with this, upon receipt of a *finishedBand[n]* signal, *NextBandCmdEnable[n]* is copied to *CmdPending[n]* and *NextBandCmdEnable[n]* is cleared. Note, each of the LBD, CDU, and TE (where present) may be re-programmed individually between bands by appropriately setting *NextBandCmdAdr[2-0]* respectively. However, execution of inter-band commands may be postponed until all blocks specified in the *BandSelectMask* register have pulsed their *finishedband* signal. This may be accomplished by only setting *NextBandCmdAdr[3]* (indirectly causing *NextBandCmdEnable[3]* to be set) in which case it is the *finishedAllBand* signal which causes *NextBandCmdEnable[3]* to be copied to *CmdPending[3]*.

To conveniently update multiple registers, for example at the start of printing a page, a series of Write Register commands can be stored in DRAM. When the start address of the first Write Register command is written to the *CmdAdr* register (via the CPU), the *CmdSource* register is automatically set to 1 to actually start the execution at *CmdAdr*.

The final instruction in the command block stored in DRAM must be a register write of 0 to *CmdSource* so that no more commands are read from DRAM. Subsequent commands will come from pending programs or can be sent via the CPU bus interface.

21.8.6.1 Debug Mode

Debug mode is implemented by reusing the normal CPU and DRAM access decode logic. When in the *Arbitrate* state (see state machine A below), the PEP address bus is defaulted to the value in the *DebugSelect* register. The top bits of the *DebugSelect* register are used to decode a select to a PEP unit and the remaining bits are reflected on the PEP address bus. The selected units read data bus is reflected on the *pcu_cpu_data* bus to the RDU in the CPU. The *pcu_cpu_debug_valid* signal indicates to the RDU that the data on the *pcu_cpu_data* bus is valid debug data. The *pcu_cpu_debug_valid* is a repeated version of the selected units ready signal *<unit>_pcu_rdy*.

Normal CPU and DRAM command access will require the PEP bus, and as such will cause the debug data to be invalid during the access, this is indicated to the RDU by setting *pcu_cpu_debug_valid* to zero.

The decode logic is :

```
// Default Debug decode
pcu_<unit>_sel          = decode(DebugSelect[15:12])
pcu_adr[11:2]           = DebugSelect[11:2]
pcu_cpu_data            = <unit>_pcu_datain[31:0]
pcu_cpu_debug_valid     = <unit>_pcu_rdy AND state == Arbitrate
```

21.8.7 State Machines

DRAM command fetching and general command execution is accomplished using two state machines. State machine A evaluates whether a CPU or DRAM command is being executed, and proceeds to execute the command(s). Since the CPU has priority over the DRAM it is permitted to interrupt the execution of a stream of DRAM commands.

Machine B decides which address should be used for DRAM access, fetches commands from DRAM and fills a command fifo which A executes. The reason for separating the two functions is to facilitate the execution of CPU or Debug commands while state machine B is performing DRAM reads and filling the command fifo. In the case where state machine A is ready to execute commands (in its *Arbitrate* state) and it sees both a full DRAM command fifo and an active *cpu_pcu_sel* then the DRAM commands are executed.

21.8.7.1 State Machine A: Arbitration and execution of commands
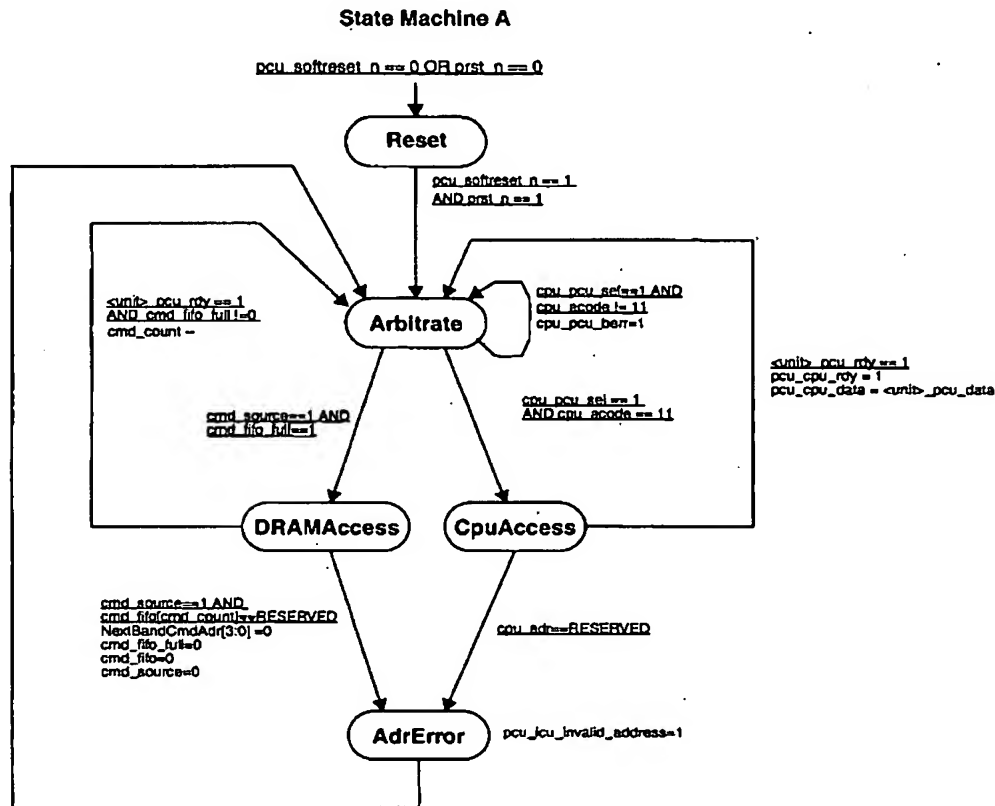
State Machine A

Figure 97. Command Arbitration and execution

The state-machine enters the *Reset* state when there is an active strobe on either the reset pin, *prst_n*, or the PCU's soft-reset register. All registers in the PCU are zeroed, unless otherwise specified, on the next rising clock edge. The PCU self-deasserts the soft reset in the *pclk* cycle after it has been asserted.

The state changes from *Reset* to *Arbitrate* when *prst_n* == 1 and *PCU_softreset* == 1.

The state-machine waits in the *Arbitrate* state until it detects a request for CPU access to the PEP units (*cpu_pcu_sel* == 1 and *cpu_acode* == 11) or a request to execute DRAM commands *CmdSource* == 1, and DRAM commands are available, *CmdFifoFull*==1. Note if (*cpu_pcu_sel* == 1 and *cpu_acode* != 11) the CPU is attempting an illegal access. The PCU ignores this command and strobes the *cpu_pcu_berr* for one cycle.

While in the *Arbitrate* state the machine assigns the *DebugSelect* register to the PCU unit decode logic and the remaining bits to the PEP address bus. When in this state the debug data returned from the selected PEP unit is reflected on the CPU bus (*pcu_cpu_data* bus) and the *pcu_cpu_debug_valid*=1.

If a CPU access request is detected (*cpu_pcu_sel* == 1 and *cpu_acode* == 11) then the machine proceeds to the *CpuAccess* state. In the *CpuAccess* state the cpu address is decoded and used to determine the PEP unit to select. The remaining address bits are passed through to the PEP address bus. The machine remains

in the *CpuAccess* state until a valid ready from the selected PEP unit is received. When received the machine returns to the arbitrate state, and the ready signal to the CPU is pulsed.

```
// decode the logic
pcu_<unit>_sel = decode(cpu_adr[15:12])
pcu_adr[11:2]  = cpu_adr[11:2]
```

If when decoding the *cpu_adr* bus, the address selects a reserved address, the state machine proceeds to the *AdrError* state, and then back to the *Arbitrate* state. An address error interrupt will be generated.

If the state machine detects a request to execute DRAM commands (*CmdSource* = 1), it will wait in the *Arbitrate* state until commands have been loaded into the command FIFO from DRAM (all controlled by state machine B). When the DRAM commands are available (*cmd_fifo_full* = 1) the state machine will proceed to the *DRAMAccess* state.

When in the *DRAMAccess* state the commands are executed from the *cmd_fifo*. A command in the *cmd_fifo* consists of 64-bits (or which the FIFO holds 4). The decoding of the 64-bits to commands is given in Table 93. For each command the decode is

```
// DRAM command decode
pcu_<unit>_sel = decode( cmd_fifo[cmd_count][15:12] )
pcu_adr[11:2]  = cmd_fifo[cmd_count][11:2]
pcu_dataout    = cmd_fifo[cmd_count][63:32]
```

When the selected PEP unit returns a ready signal (*<unit>_pcu_rdy*=1) indicating the command has completed, the state machine will return to the *Arbitrate* state. If more commands exists (*cmd_count* !=0) the transition will decrement the command count.

When in the *DRAMAccess* state, if when decoding the DRAM command address bus (*cmd_fifo[cmd_count][15:12]*), the address selects a reserved address, the state machine proceeds to the *AdrError* state, and then back to the *Arbitrate* state. An address error interrupt will be generated and the DRAM command FIFOs will be cleared.

A CPU access can pre-empt any pending DRAM commands. After each command is completed the state machine returns to the *Arbitrate* state. If a CPU and DRAM command are pending the CPU command always takes priority. If a CPU or DRAM command sets the *CmdSource* to 0, all subsequent DRAM commands in the command FIFO are cleared. If the CPU sets the *CmdSource* to 0 the *CmdPending* and *NextBandCmdEnable* work registers are also cleared.

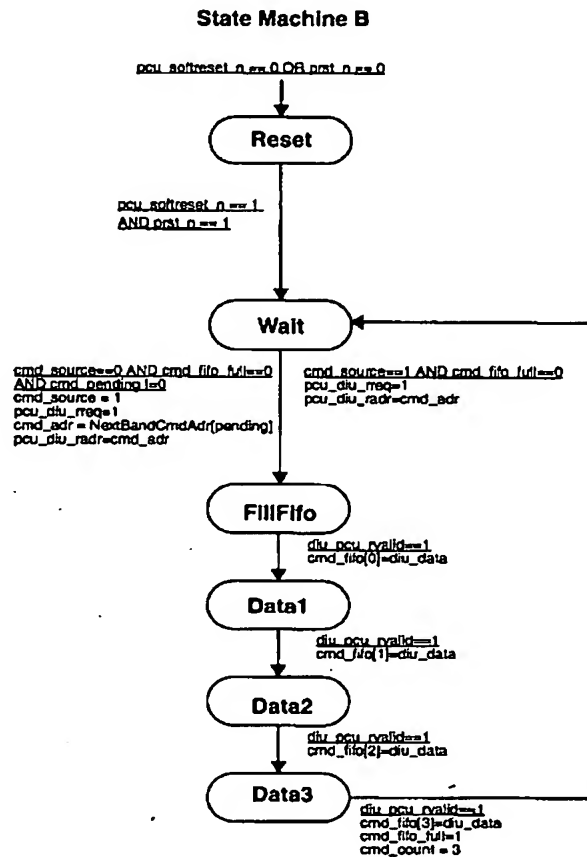### 21.8.7.2 State Machine B: Fetching DRAM commands

State Machine B

pcu_softreset_n == 0 OR prst_n == 0

Figure 98. DRAM command access state machine

A system reset ($prst\_n$==0) or a software reset ($pcu\_softreset\_n$==0) will cause the state machine to reset to the *Reset* state. The state machine remains in the *Reset* until both reset conditions are removed. When removed the machine proceeds to the *Wait* state.

The state machine waits in the *Wait* state until it determines that commands are needed from DRAM. Two possible conditions exist that require DRAM access. Either the PCU is processing commands which must be fetched from DRAM (cmd_source==1), and the command FIFO is empty (cmd_fifo_full==0), or the command FIFO is empty and there are some commands pending ($cmd\_pending$!=0). In either of these conditions the machine proceeds to the *FillFifo* state and issues a read request to DRAM (pcu_diu_rreq==1), it calculates the address to read from dependent on the transition condition. In the command pending transition condition, the highest priority *NextBandCmdAdr* that is pending is used for the read address (pcu_diu_radr) and is also copied to the *CmdAdr* register. In the normal PCU processing transition the pcu_diu_radr is the *CmdAdr* register.

In the *FillFifo* state the machine wait for the DRAM to respond to the read request and transfer data words. On receipt of the first word of data *diu_pcu_rvalid*=1, the machine stores the 64-bit data word in the command FIFO (*cmd_fifo[0]*) and transitions to the *Data1*, *Data2*, *Data3* states each time waiting for a *diu_pcu_rvalid*=1 and storing the transferred data word to *cmd_fifo[1]*, *cmd_fifo[2]* and *cmd_fifo[3]* respectively.

When the transfer is complete the machine returns to the *Wait* state, setting the *cmd_count* to 3 and the *cmd_fifo_full*=1.

21.8.7.3 PCU_ICU_Address_Invalid Interrupt

When the PCU is executing commands, addresses decoded from commands which are not PCU mapped addresses (4-bits only) will result in the current command being ignored and the *pcu_invalid_address* interrupt signal is strobed. If this command is from DRAM all remaining commands already retrieved from DRAM are flushed from the *CmdFifo, CmdPending, NextBandCmdEnable* and *CmdSource* are cleared to zero.) These registers are uneffected if the command is from the CPU. The CPU can then interrogate the PCU to find the source of the illegal command via the *InvalidAddress* register.

22 Contone Decoder Unit (CDU)

22.1 OVERVIEW

The Contone Decoder Unit (CDU) is responsible for performing the optional decompression of the contone data layer.

The input to the CDU is up to 4 planes of compressed contone data in JPEG interleaved format. This will typically be 3 planes, representing a CMY contone image, or 4 planes representing a CMYK contone image. The CDU must support a page of A4 length (11.7 inches) and Letter width (8.5 inches) at a resolution of 267 ppi in 4 colors and a print speed of 1 side per 2 seconds.

The CDU and the other page expansion units support the notion of page banding. A compressed page is divided into one or more bands, with a number of bands stored in memory. As a band of the page is consumed for printing a new band can be downloaded. The new band may be for the current page or the next page. Band-finish interrupts have been provided to notify the CPU of free buffer space.

The compressed contone data is read from the on-chip DRAM. The output of the CDU is the decompressed contone data, separated into planes. The decompressed contone image is written to a circular buffer in DRAM with an expected minimum size of 12 lines and a configurable maximum. The decompressed contone image is subsequently read a line at a time by the CFU, optionally color converted, scaled up to 1600 ppi and then passed on to the HCU for the next stage in the printing pipeline. The CDU also outputs a *cdu_finishedband* control flag indicating that the CDU has finished reading a band of compressed contone data in DRAM and that area of DRAM is now free. This flag is used by the PCU and is available as an interrupt to the CPU.

22.2 STORAGE REQUIREMENTS FOR DECOMPRESSED CONTONE DATA IN DRAM

A single SoPEC must support a page of A4 length (11.7 inches) and Letter width (8.5 inches) at a resolution of 267 ppi in 4 colors and a print speed of 1 side per 2 seconds. The printheads specified in the Bi-lithic Printhead Specification [2] have 13824 nozzles per color to provide full bleed printing for A4 and Letter. At 267 ppi, there are 2304 contone pixels[1] per line represented by 288 JPEG blocks per color. However each of these blocks actually stores data for 8 lines, since a single JPEG block is 8 x 8 pixels. The CDU produces contone data for 8 lines in parallel, while the HCU processes data linearly across a line on a line by line basis. The contone data is decoded only once and then buffered in DRAM. This means we require two sets of 8 buffer-lines - one set of 8 buffer lines is being consumed by the CFU while the other set of 8 buffer lines is being generated by the CDU.

The buffer requirement can be reduced by using a 1.5 buffering scheme, where the CDU fills 8 lines while the CFU consumes 4 lines. The buffer space required is a minimum of 12 line stores per color, for a total space of 108 KBytes[2]. A circular buffer scheme is employed whereby the CDU may only begin to write a line of JPEG blocks (equals 8 lines of contone data) when there are 8-lines free in the buffer. Once the full 8 lines have been written by the CDU, the CFU may now begin to read them on a line by line basis.

This reduction in buffering comes with the cost of an increased peak bandwidth requirement for the CDU write access to DRAM. The CDU must be able to write the decompressed contone at twice the rate at which the CFU reads the data. To allow for trade-offs to be made between peak bandwidth and amount of storage, the size of the circular buffer is configurable. For example, if the circular buffer is configured to be 16 lines it behaves like a double-buffer scheme where the peak bandwidth requirements of the CDU and

---

1. Pixels may be 8, 16, 24 or 32 bits depending on the number of color planes (8-bits per color)
2. 12 lines x 4 colors x 2304 bytes (assumes 267 ppi, 4 color, full bleed A4/Letter)

CFU are equal. An increase over 16 lines allows the CDU to write ahead of the CFU and provides it with a margin to cope with very poor local compression ratios in the image.

SoPEC should also provide support for A3 printing and printing at resolutions above 267 ppi. This increases the storage requirement for the decompressed contone data (buffer) in DRAM. Table 94 gives the storage requirements for the decompressed contone data at some sample contone resolutions for different page sizes. It assumes 4 color planes of contone data and a 1.5 buffering scheme.

Table 94. Storage requirements for decompressed contone data (buffer)

Page area	Contone resolution (ppi)	Scale factor	Pixels per line	Storage required (KBytes)
A4/Letter[b]	267	6	2304	108[d]
	400	4	3456	162
	800	2	6912	324
A3[c]	267	6	3248	152.25
	400	4	4872	228.37
	800	2	9744	456.75

a. Required for CFU to convert to final output at 1600 dpi

b. Bi-lithic printhead has 13824 nozzles per color providing full bleed printing for A4/Letter

c. Bi-lithic printhead has 19488 nozzles per color providing full bleed printing for A3

d. 12 lines x 4 colors x 2304 bytes.

22.3 DECOMPRESSION PERFORMANCE REQUIREMENTS

The JPEG decoder core can produce a single color pixel every system clock (*pclk*) cycle, making it capable of decoding at a peak output rate of 8 bits/cycle. SoPEC processes 1 dot (bi-level in 6 colors) per system clock cycle to achieve a print speed of 1 side per 2 seconds for full bleed A4/Letter printing. The CFU replicates pixels a scale factor (SF) number of times in both the horizontal and vertical directions to convert the final output to 1600 ppi. Thus the CFU consumes a 4 color pixel (32 bits) every SF x SF cycles. The 1.5 buffering scheme described in section 22.2 on page 265 means that the CDU must write the data at twice this rate. With support for 4 colors at 267 ppi, the decompression output bandwidth requirement is 1.78 bits/cycle[1].

The JPEG decoder is fed directly from the main memory via the DRAM interface. The amount of compression determines the input bandwidth requirements for the CDU. As the level of compression increases, the bandwidth decreases, but the quality of the final output image can also decrease. Although the average compression ratio for contone data is expected to be 10:1, the average bandwidth allocated to the CDU allows for a local minimum compression ratio of 5:1 over a single line of JPEG blocks. This equates to a peak input bandwidth requirement of 0.36 bits/cycle for 4 colors at 267 ppi, full bleed A4/Letter printing at 1 side per 2 seconds.

Table 95 gives the decompression output bandwidth requirements for different resolutions of contone data to meet a print speed of 1 side per 2 seconds. Higher resolution requires higher bandwidth and larger storage for decompressed contone data in DRAM. A resolution of 400 ppi contone data in 4 colors requires 4 bits/cycle[2], which is practical using a 1.5 buffering scheme. However, a resolution of 800 ppi would require a double buffering scheme (16 lines) so the CDU only has to match the CFU consumption rate. In

1. 2 x ((4 colors x 8 bits) / (6 x 6 cycles)) = 1.78 bits/cycle
2. 2 x ((4 colors x 8 bits) / (4 x 4 cycles)) = 4 bits/cycle

this case the decompression output bandwidth requirement is **8 bits/cycle**[1], the limiting factor being the output rate of the JPEG decoder core.

Table 95. CDU performance requirements for full bleed A4/Letter printing at 1 side per 2 seconds.

Contone resolution (ppi)	Scale factor	Decompression output bandwidth requirement (bits/cycle)
267	6	1.78
400	4	4
800	2	8[b]

a. Assumes 4 color pixel contone data and a 12 line buffer.

b. Scale factor 2 requires at least a 16 line buffer.

22.4   DATA FLOW

Figure 99 shows the general data flow for contone data - compressed contone planes are read from DRAM by the CDU, and the decompressed contone data is written to the 12-line circular buffer in DRAM. The line buffers are subsequently read by the CFU.
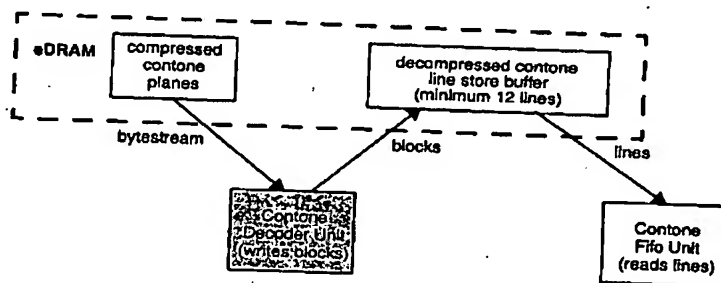
Figure 99. Outline of contone data flow with respect to CDU

The CDU allows the contone data to be passed directly on, which will be the case if the color represented by each color plane in the JPEG image is an available ink. For example, the four colors may be C, M, Y, and K, directly represented by CMYK inks. The four colors may represent gold, metallic green etc. for multi-SoPEC printing with exact colors.

However JPEG produces better compression ratios for a given visible quality when luminance and chrominance channels are separated. With CMYK, K can be considered to be luminance, but C, M, and Y each contain luminance information, and so would need to be compressed with appropriate luminance tables. We therefore provide the means by which CMY can be passed to SoPEC as YCrCb. K does not need color conversion. When being JPEG compressed, CMY is typically converted to RGB, then to YCrCb and then finally JPEG compressed. At decompression, the YCrCb data is obtained and written to the decompressed contone store by the CDU. This is read by the CFU where the YCrCb can then be optionally color converted to RGB, and finally back to CMY.

---

1. (4 colors x 8 bits) / (2 x 2 cycles) = 8 bits/cycle

The external RIP provides conversion from RGB to YCrCb, specifically to match the actual hardware implementation of the inverse transform within SoPEC, as per CCIR 601-2 [20] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding.

The CFU provides the translation to either RGB or CMY. RGB is included since it is a necessary step to produce CMY, and some printers increase their color gamut by including RGB inks as well as CMYK.

22.5   IMPLEMENTATION

A block diagram of the CDU is shown in Figure 100.
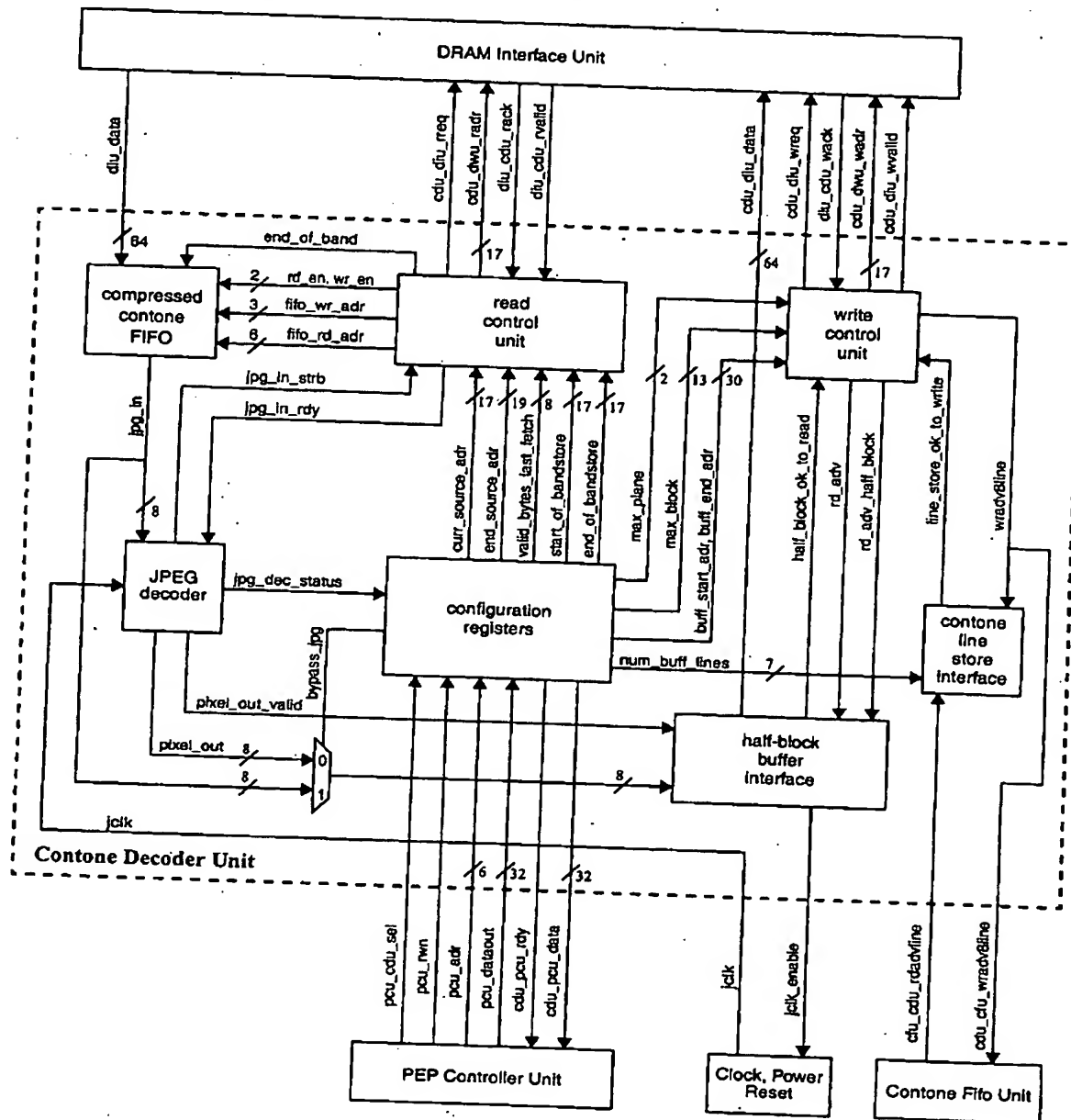
Figure 100. Block diagram of CDU

All output signals from the CDU (*cdu_cfu_wradv8line*, *cdu_finishedband*, *cdu_icu_jpegerror*, and control signals to the DIU) must always be valid after reset. If the CDU is not currently decoding, *cdu_cfu_wradv8line*, *cdu_finishedband* and *cdu_icu_jpegerror* will always be 0.

The read control unit is responsible for keeping the JPEG decoder's input FIFO full by reading compressed contone bytestream from external DRAM via the DIU, and produces the *cdu_finishedband* signal. The write control unit accepts the output from the JPEG decoder a half JPEG block (32 bytes) at a time, writes it into a double-buffer, and writes the double buffered decompressed half blocks to DRAM via the DIU, interacting with the CFU in order to share DRAM buffers.

22.5.1 Definitions of I/O

Table 96. CDU port list and description

Port name	Pins	I/O	Description
Clocks and reset			
pclk	1	In	System clock.
jclk	1	In	Gated version of system clock used to clock the JPEG decoder core and logic at the output of the core. Allows for stalling of the JPEG core at a pixel sample boundary.
jclk_enable	1	Out	Gating signal for jclk.
prst_n	1	In	System reset, synchronous active low.
jrst_n	1	In	Reset for jclk domain, synchronous active low.
PCU interface			
pcu_cdu_sel	1	In	Block select from the PCU. When *pcu_cdu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
cdu_pcu_rdy	1	Out	Ready signal to the PCU. When *cdu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *cdu_pcu_data* is valid.
cdu_pcu_data[31:0]	32	Out	Read data bus to the PCU.
DIU read interface			
cdu_diu_rreq	1	Out	CDU read request, active high. A read request must be accompanied by a valid read address.
diu_cdu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *cdu_diu_radr*.
cdu_diu_radr[21:5]	17	Out	CDU read address. 17 bits wide (256-bit aligned word).
diu_cdu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DRAM.
DIU write interface			
cdu_diu_wreq	1	Out	CDU write request, active high. A write request must be accompanied by a valid write address and valid write data.
diu_cdu_wack	1	In	Acknowledge from DIU, active high. Indicates that a write request has been accepted and the new write address can be placed on the address bus, *cdu_diu_wadr*.

Table 96. CDU port list and description

Port name	# pins	I/O	Description
cdu_diu_wadr[21:3]	19	Out	CDU write address. 19 bits wide (64-bit aligned word).
cdu_diu_wvalid	1	Out	Write data valid, active high. Indicates that valid data is now on the write data bus, cdu_diu_data.
cdu_diu_data[63:0]	64	Out	Write data bus.
CFU Interface			
cfu_cdu_rdadvline	1	In	Read line pulse, active high. Indicates that the CFU has finished reading a line of decompressed contone data to the circular buffer in DRAM and that line of the buffer is now free.
cdu_cfu_wradv8line	1	Out	Write 8line pulse, active high. Indicates that the CDU has finished writing to 8 lines of decompressed contone data to the circular buffer in DRAM and the data is available to be read by the CFU.
TE and LBD Interface			
cdu_start_of_bandstore[21:5]	17	Out	Points to the 256-bit word that defines the start of the memory area allocated for page bands.
cdu_end_of_bandstore[21:5]	17	Out	Points to the 256-bit word that defines the last address of the memory area allocated for page bands.
ICU Interface			
cdu_finishedband	1	Out	CDU's finishedBand flag, active high. Interrupt to the CPU to indicate that the CDU has finished processing a band of compressed contone data in DRAM and that area of DRAM is now free.This signal goes to both the interrupt controller and the PCU.
cdu_icu_jpegerror	1	Out	Active high interrupt indicating an error has occurred in the JPEG decoding process and decompression has stopped. A reset of the CDU must be performed to clear this interrupt.

22.5.2 Configuration registers

The configuration registers in the CDU are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for the description of the protocol and timing diagrams for reading and writing registers in the CDU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the CDU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cdu_pcu_data*.

Since the CDU, LBD and TE all access the page band store, they share two registers that enable sequential memory accesses to the page band stores to be circular in nature. Table 97 lists these two registers.

Table 97. Registers shared between the CDU, LBD, and TE

Address (CDU base)	Register name	#bits	Value on reset	Description
Setup registers (remain constant during the processing of multiple bands)				
0x80	StartOfBandStore	17	0x0_0000	Points to the 256-bit word that defines the start of the memory area allocated for page bands. Circular address generation wraps to this start address.
0x84	EndOfBandStore	17	0x1_FFFF	Points to the 256-bit word that defines the last address of the memory area allocated for page bands. If the current read address is from this address, then instead of adding 1 to the current address, the current address will be loaded from the StartOfBandStore register.

The CDU contains the following additional registers:

Table 98. CDU registers

Address (CDU base)	Register name	#bits	Value on reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the CDU. This terminates all internal operations within the CS6150. All configuration data previously loaded into the core except for the tables is deleted.
0x04	Go	1	0x0	Writing 1 to this register starts the CDU. Writing 0 to this register halts the CDU. When Go is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When Go is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). NextBandEnable is cleared when Go is asserted. The CFU must be started before the CDU is started. This register can be read to determine if the CDU is running (1 - running, 0 - stopped).
Setup registers				
0x10	MaxPlane	2	0x0	Defines the number of contone planes - 1. For example, this will be 0 for K (greyscale printing), 2 for CMY, and 3 for CMYK.

Table 98. CDU registers

Address (CDU base)	Register name	#bits	Value on reset	Description
0x14	MaxBlock	13	0x000	Number of JPEG MCUs (or JPEG block equivalents, i.e. 8x8 bytes) in a line - 1.
0x18	BuffStartAdr	15	0x0000	Points to the start of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x1C	BuffEndAdr	15	0x0000	Points to the start of the last half JPEG block at the end of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x20	NumBuffLines	7	0x0C	Defines size of buffer in DRAM in terms of the number of decompressed contone lines. The size of the buffer should be a multiple of 4 lines with a minimum size of 8 lines.
0x24	BypassJpg	1	0x0	Determines whether or not the JPEG decoder will be bypassed (and hence pixels are copied directly from input to output) 0 - don't bypass, 1 - bypass Should not be changed between bands.
0x30	NextBandCurrSourceAdr	17	0x0_0000	The 256-bit aligned word address containing the start of the next band of compressed contone data in DRAM. This value is copied to CurrSourceAdr when both DoneBand is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0x34	NextBandEndSourceAdr	19	0x0_0000	The 64-bit aligned word address containing the last bytes of the next band of compressed contone data in DRAM. This value is copied to EndSourceAdr when when both DoneBand is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0x38	NextBandValidBytesLastFetch	8	0x00	Mask containing a 1 in each bit position that represents a valid byte in the last 64-bit fetch of the next band of compressed contone data from DRAM. This value is copied to ValidBytesLastFetch when both DoneBand is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0x3C	NextBandEnable	1	0x0	When NextBandEnable is 1 and DoneBand is 1, then when cdu_finishedband is set at the end of a band -NextBandCurrSourceAdr is copied to CurrSourceAdr, -NextBandEndSourceAdr is copied to EndSourceAdr -NextBandValidBytesLastFetch is copied to ValidBytesLastFetch -DoneBand is cleared, -NextBandEnable is cleared. NextBandEnable is cleared when Go is asserted.
Read-only registers				

Table 98. CDU registers

Address CDU base	Register name	bits	Value on reset	Description
0x40	DoneBand	1	0x0	Specifies whether or not the current band has finished loading into the local FIFO. It is cleared to 0 when *Go* transitions from 0 to 1. When the last of the compressed contone data for the band has been loaded into the local FIFO, the *cdu_finishedband* signal is given out and the *DoneBand* flag is set. If *NextBandEnable* is 1 at this time then *CurrSourceAdr*, *EndSourceAdr* and *ValidBytesLastFetch* are updated with the values for the next band and *DoneBand* is cleared. Processing of the next band starts immediately. If *NextBandEnable* is 0 then the remainder of the CDU will continue to run, decompressing the data already loaded, while the read control unit waits for *NextBandEnable* to be set before it restarts.
0x44	CurrSourceAdr	17	0x0_0000	The current 256-bit aligned word address within the current band of compressed contone data in DRAM.
0x48	EndSourceAdr	19	0x0_0000	The 64-bit aligned word address containing the last bytes of the current band of compressed contone data in DRAM.
0x4C	ValidBytesLast-Fetch	8	0x00	Mask containing a 1 in each bit position that represents a valid byte in the fast 64-bit fetch of the current band of compressed contone data from DRAM. If the lower 3 bytes are valid, then the lower 3 bits of *ValidBytesLastFetch* should be set and the upper 5 bits should be clear.
JPEG decoder core setup registers				
0x50	JpgDecMask	5	0x00	As segments are decoded they can also be output on the *DecJpg* (*JpgDecHdr*) port with the user selecting the segments for output by setting bits in the JpgDecMask port as follows: 4 SOF+SOS+DNL 3 COM+APP 2 DRI 1 DQT 0 DHT If any one of the bits of jpgDecMask is asserted then the SOI and EOI markers are also passed to the *DecJpg* port.
0x54	JpgDecTType	1	0x0	Test type selector: 0 - DCT coefficients displayed on *JpgDecTdata* 1 - QDCT coefficient displayed on *JpgDecTdata*
0x58	JpgDecTestEn	1	0x0	Signal which causes the memories to be bypassed for test purposes.
0x5C	JpgDecPType	4	0x0	Signal specifying parameters to be placed on port *JpgDecPValue* (See Table 99).
JPEG decoder core read-only status registers				
0x60	JpgDecHdr	8	0x00	Selected header segments from the JPEG stream that is currently being decoded. Segments selected using *JpgMask*.

Table 98. CDU registers

Address (CDU base+)	Register name	#bits	Value on reset	Description
0x64	JpgDecTData	13	0x0000	13 - TSOS output of CS1650, indicates the first output byte of the first 8x8 block of the test data. 12 - TSOB output of CS1650, indicates the first output byte of each 8x8 block of test data. 11-0 - 11-bit output test data port - displays DCT coefficients or quantized coefficients depending on value of *JpgDecTType*.
0x68	JpgDecPValue	16	0x0000	Decoding parameter bus which enables various parameters used by the core to be read. The data available on the PValue port is for information only, and does not contain control signals for the decoder core.
0x6C	JpgDecStatus	22	0x00_0000	Bit 21 - *jpg_core_stall* (if set, indicates that the JPEG core is stalled by gating of jclk as the output JPEG halfblock double-buffers of the CDU are full) Bit 20 - *pix_out_valid* (This signal is an output from the JPEG decoder core and is asserted when a pixel is being output Bits 19-16 - *fifo_contents* (FIFO at input of JPEG decoder core) Bits 15-0 are JPEG decoder status outputs from the CS6150 (see Table 100 for description of bits).

22.5.3 Typical operation

The CDU should only be started after the CFU has been started.

For the first band of data, users set up *NextBandCurrSourceAdr*, *NextBandEndSourceAdr*, *NextBandValidBytesLastFetch*, and the various *MaxPlane*, *MaxBlock*, *BuffStartBlockAdr*, *BuffEndBlockAdr* and *NumBuffLines*. Users then set the CDU's *Go* bit to start processing of the band. When the compressed contone data for the band has finished being read in, the *cdu_finishedband* interrupt will be sent to the PCU and CPU indicating that the memory associated with the first band is now free. Processing can now start on the next band of contone data.

In order to process the next band *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and *NextBandValidBytesLastFetch* need to be updated before finally writing a 1 to *NextBandEnable*. There are 4 mechanisms for restarting the CDU between bands:

    a. *cdu_finishedband* causes an interrupt to the CPU. The CDU will have set its *DoneBand* bit. The CPU reprograms the *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and *NextBandValidBytesLastFetch* registers, and sets *NextBandEnable* to restart the CDU.

    b. The CPU programs the CDU's *NextBandCurrSourceAdr*, *NextBandCurrEndAdr* and *NextBandValidBytesLastFetch* registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the CDU sets *DoneBand*. As *NextBandEnable* is already 1, the CDU starts processing the next band immediately.

    c. The PCU is programmed so that *cdu_finishedband* triggers the PCU to execute commands from DRAM to reprogram the *NextBandCurrSourceAdr*, *NextBandEndSourceAdr* and *NextBandValidBytesLastFetch* registers and set the *NextBandEnable* bit to start the CDU processing the next band. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

    d. This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the CDU's *NextBandCurrSourceAdr*, *NextBandCurrEndAdr* and *NextBandValidBytesLastFetch*

registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the CDU sets *DoneBand* and pulses *cdu_finishedband*. As *NextBandEnable* is already 1, the CDU starts processing the next band immediately. Simultaneously, *cdu_finishedband* triggers the PCU to fetch commands from DRAM. The CDU will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the CDU's next band shadow registers and sets the *NextBandEnable* bit.

If an error occurs in the JPEG stream, the JPEG decoder will suspend its operation, an error bit will be set in the *JpgDecStatus* register and the core will ignore any input data and await a reset before starting decoding again. An interrupt is sent to the CPU by asserting *cdu_icu_jpegerror* and the CDU should then be reset by means of a write to its *Reset* register before a new page can be printed.

22.5.4 Read control unit

The read control unit is responsible for reading the compressed contone data and passing it to the JPEG decoder via the FIFO. The compressed contone data is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 208. Read accesses to DRAM are implemented by means of the state machine described in Figure 101.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *DoneBand* bit to tell it whether to attempt to read a band of compressed contone data. When *DoneBand* is set, the state machine does nothing. When *DoneBand* is clear, the state machine continues to load data into the JPEG input FIFO up to 256-bits at a time while there is space available in the FIFO. Note that the state machine has no knowledge about numbers of blocks or numbers of color planes - it merely keeps the JPEG input FIFO full by consecutive reads from DRAM. The DIU is responsible for ensuring that DRAM requests are satisfied at least at the peak DRAM read bandwidth of 0.36 bits/cycle (see section 22.3 on page 266).

A modulo 4 counter, *rd_count*, is use to count each of the 64-bits received in a 256-bit read access. It is incremented whenever *diu_cdu_rvalid* is asserted. As each 64-bit value is returned, indicated by *diu_cdu_rvalid* being asserted, *curr_source_adr* is compared to both *end_source_adr* and *end_of_bandstore*:

- If {*curr_source_adr,rd_count*} equals *end_source_adr*, the *end_of_band* control signal sent to the FIFO is 1 (to signify the end of the band), the *finishedCDUBand* signal is output, and the *DoneBand* bit is set. The remaining 64-bit values in the burst from the DIU are ignored, i.e. they are not written into the FIFO.

- If *rd_count* equals 3 and {*curr_source_adr,rd_count*} does not equal *end_source_adr*, then *curr_source_adr* is updated to be either *start_of_bandstore* or *curr_source_adr* + 1, depending on whether *curr_source_adr* also equals *end_of_bandstore*. The *end_of_band* control signal sent to the FIFO is 0.

curr_source_adr is output to the DIU as *cdu_diu_radr*.

A count is kept of the number of 64-bit values in the FIFO. When *diu_cdu_rvalid* is 1 and *ignore_data* is 0, data is written to the FIFO by asserting *FifoWr*, and *fifo_contents[3:0]* and *fifo_wr_adr[2:0]* are both incremented.

When *fifo_contents[3:0]* is greater than 0, *jpg_in_strb* is asserted to indicate that there is data available in the FIFO for the JPEG decoder core. The JPEG decoder core asserts *jpg_in_rdy* when it is ready to receive data from the FIFO. Note it is also possible to bypass the JPEG decoder core by setting the *BypassJpg* register to 1. In this case data is sent directly from the FIFO to the half-block double-buffer. While the JPEG decoder is not stalled (*jpg_core_stall* equal 0), and *jpg_in_rdy* (or *bypass_jpg*) and *jpg_in_strb* are both 1, a byte of data is consumed by the JPEG decoder core. *fifo_rd_adr[5:0]* is then incremented to select the next byte. The read address is byte aligned, i.e. the upper 3 bits are input as the read address for the FIFO

and the lower 3 bits are used to select a byte from the 64 bits. If *fifo_rd_adr[2:0]* = 111 then the next 64-bit value is read from the FIFO by asserting *fifo_rd*, and *fifo_contents[3:0]* is decremented.
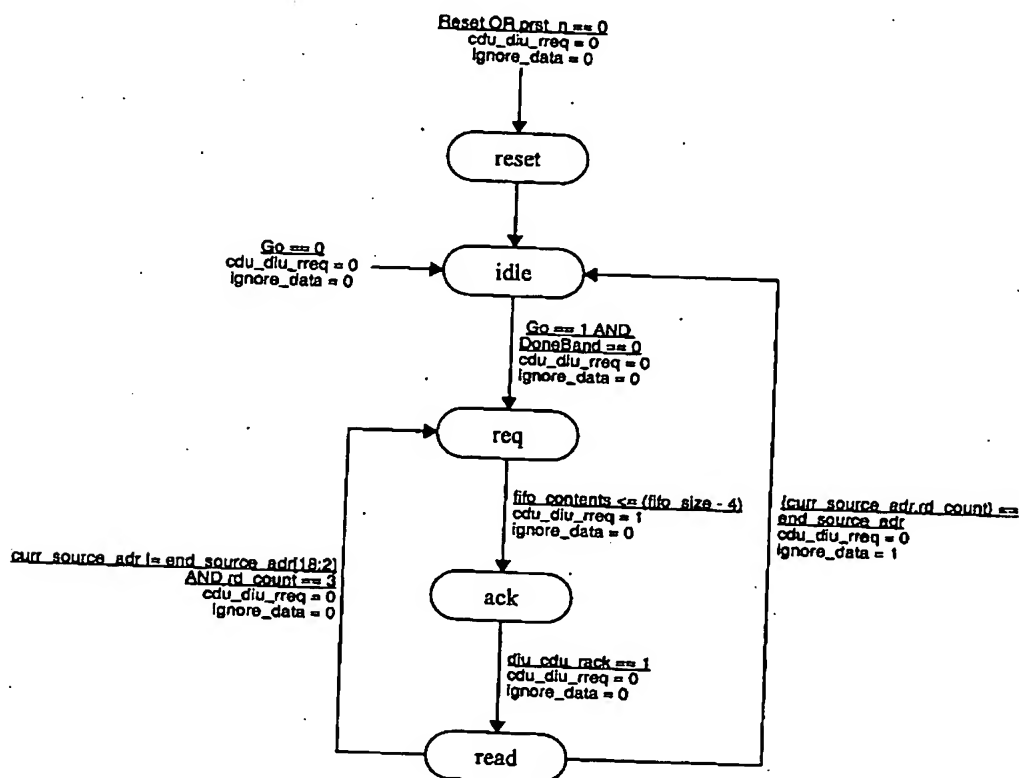

Figure 101. State machine to read compressed contone data

22.5.5 Compressed contone FIFO

The compressed contone FIFO conceptually is a 64-bit input, and 8-bit output FIFO to account for the 64-bit data transfers from the DIU, and the 8-bit requirement of the JPEG decoder.

In reality, the FIFO is actually 8 entries deep and 65-bits wide (to accommodate two 256-bit accesses), with bits 63-0 carrying data, and bit 64 containing a 1-bit *end_of_band* flag. Whenever 64-bit data is written to the FIFO from the DIU, an *end_of_band* flag is also passed in from the read control unit. The *end_of_band* bit is 1 if this is the last data transfer for the current band, and 0 if it is not the last transfer. When *end_of_band* = 1 during an input, the *ValidBytesLastFetch* register is also copied to an image version of the same.

On the JPEG decoder side of the FIFO, the read address is byte aligned, i.e. the upper 3 bits are input as the read address for the FIFO and the lower 3 bits are used to select a byte from the 64 bits (1st byte corresponds to bits 7-0, second byte to bits 15-8 etc.). If bit 64 is set on the read, bits 63-0 contain the end of the

bytestream for that band, and only the bytes specified by the image of *ValidBytesLastFetch* are valid bytes to be read and presented to the JPEG decoder.

Note that *ValidBytesLastFetch* is copied to an image register as it may be possible for the CDU to be reprogrammed for the next band before the previous band's compressed contone data has been read from the FIFO (as an additional effect of this, the CDU has a non-problematic limitation in that each band of contone data must be more than 4 × 64-bits, or 32 bytes, in length).

22.5.6   CS6150 JPEG decoder

JPEG decoder functionality is implemented by means of a modified version of the Amphion CS6150 JPEG decoder core. The decoder is run at a nominal clock speed of 160 MHz. (Amphion have stated that the CS6150 JPEG decoder core can run at 185 MHz in 0.13um technology). The core is clocked by *jclk* which a gated version of the system clock pclk. Gating the clock provides a mechanism for stalling the JPEG decoder on a single color pixel-by-pixel basis. Control of the flow of output data is also provided by the *PixOutEnab* input to the JPEG decoder. However, this only allows stalling of the output at a JPEG block boundary and is insufficient for SoPEC. Thus gating of the clock is employed and *PixOutEnab* is instead tied high.

The CS6150 decoder automatically extracts all relevant parameters from the JPEG bytestream and uses them to control the decoding of the image. The JPEG bytestream contains data for the Huffman tables, quantization tables, restart interval definition and frame and scan headers. The decoder parses and checks the JPEG bytestream automatically detecting and processing all the JPEG marker segments. After identifying the JPEG segments the decoder re-directs the data to the appropriate units to be stored or processed as appropriate. Any errors detected in the bytestream, apart from those in the entropy coded segments, are signalled and, if an error is found, the decoder stops reading the JPEG stream and waits to be reset.

JPEG images must have their data stored in interleaved format with no subsampling. Images longer than 65536 lines are allowed: these must have an initial imageHeight of 0. If the image has a Define Number Lines (DNL) marker at the end (normally necessary for standard JPEG, but not necessary for SoPEC's version of the CS6150), it must be equal to the total image height mod 64k or an error will be generated.

See the CS6150 Databook [17] for more details on how the core is used, and for timing diagrams of the interfaces. Note that [17] does not describe the use of the DNL marker in images of more than 64k lines length as this is a modification to the core.

The CS6150 decoder can be bypassed by setting the *BypassJpg* register. If this register is set, then the data read from DRAM must be in the same format as if it was produced by the JPEG decoder: 8x8 blocks of pixels in the correct color order. The data is uncompressed and is therefore lossless.

The following subsections describe the means by which the CS6150 internals can be made visible.

22.5.6.1   JPEG decoder parameter bus

The decoding parameter bus *JpgDecPValue* is a 16-bit port used to output various parameters extracted from the input data stream and currently used by the core. The 4-bit selector input (*JpgDecPType*) determines which internal parameters are displayed on the parameter bus as per Table 99. The data available on the *PValue* port does not contain control signals used by the CS6150.

Table 99. Parameter bus definitions

PType	Output orientation	PValue
0x0	FY[15:0]	FY: number of lines in frame
0x1	FX[15:0]	FX: number of columns in frame
0x2	00_YMCU[13:0]	YMCU: number of MCUs in Y direction of the current scan

Table 99. Parameter bus definitions

PType	Output orientation	PValue
0x3	00_XMCU[13:0]	XMCU: number of MCUs in X direction of the current scan
0x4	Cs0[7:0]_Tq0[1:0]_V0[2:0]_H0[2:0]	Cs0: identifier for the first scan component Tq0: quantization table identifier for the first scan component V0: vertical sampling factor for the first scan component. Values = 1-4 H0: horizontal sampling factor for the first scan component. Values = 1-4
0x5	Cs1[7:0]_Tq1[1:0]_V1[2:0]_H1[2:0]	Cs1, Tq1, V1 and H1 for the second scan component. V1, H1 undefined if NS<2
0x6	Cs2[7:0]_Tq2[1:0]_V2[2:0]_H2[2:0]	Cs2, Tq2, V2 and H2 for the second scan component. V2, H2 undefined if NS<3
0x7	Cs3[7:0]_Tq3[1:0]_V3[2:0]_H3[2:0]	Cs3, Tq3, V3 and H3 for the second scan component. V3, H3 undefined if NS<4
0x8	CsH[15:0]	CsH: no. of rows in current scan
0x9	CsV[15:0]	CsV: no. of columns in current scan
0xA	DRI[15:0]	DRI: restart interval
0xB	000_HMAX[2:0]_VMAX[2:0]_MCUBLK[3:0]_NS[2:0]	HMAX: maximal horizontal sampling factor in frame VMAX: maximal vertical sampling factor in frame MCUBLK: number of blocks per MCU of the current scan, from 1 to 10 NS: number of scan components in current scan, 1-4

22.5.6.2 JPEG decoder status register

The status register flags indicate the current state of the CS6150 operation. When an error is detected during the decoding process, the decompression process in the JPEG decoder is suspended and an interrupt is sent to the CPU by asserting *cdu_icu_jpegerror* (generated by the ORing of *CtlError*, *HtError*, *QtError* and *DecError*). Go is also cleared to halt the CDU. The CPU can check the source of the error by reading the *JpgDecStatus register*. The CS6150 waits until a reset process is invoked by asserting the hard reset *prst_n* or by a soft reset of the CDU. The individual bits of *JpgDecStatus* are set to zero at reset and active high to indicate an error condition as defined in Table 100.

Note: A *DecHfError* will not block the input as the core will try to recover and produce the correct amount of pixel data. The *DecHfError* is cleared automatically at the start of the next image and so no intervention is required from the user. If any of the other errors occur in the decode mode then, following the error cancellation, the core will discard all input data until the next Start Of Image (SOI) without triggering any more errors.

The progress of the decoding can be monitored by observing the values of *TblDef*, *IDctInProg*, *DecInProg* and *JpgInProg*.

Table 100. JPEG decoder status register definitions

Bit	Name	Description
15 - 12	TblDef[7:4]	Indicates the number of Huffman tables defined, 1bit/table.
11 - 8	TblDef[3:0]	Indicates the number of quantization tables defined, 1bit/table.
7	DecHfError	Set when an undefined Huffman table symbol is referenced during decoding.

Table 100. JPEG decoder status register definitions

Bit	Name	Description
6	CtlError	Set when an invalid SOF parameter or an invalid SOS parameter is detected. Also set when there is a mismatch between the DNL segment input to the core and the number of lines in the input image which have already been decoded. Note that SoPEC's implementation of the CS6150 does not require a final DNL when the initial setting for ImageHeight is 0. This is to allow images longer than 64k lines.
5	HtError	Set when an invalid DHT segment is detected.
4	QtError	Set when an invalid DQT segment is detected.
3	DecError	Set when anything other than a JPEG marker is input. Set when any of DecFlags[6:4] are set. Set when any data other than the SOI marker is detected at the start of a stream. Set when any SOF marker is detected other than SOF0. Set if incomplete Huffman or quantization definition is detected.
2	IDctInProg	Set when IDCT starts processing first data of a scan. Cleared when IDCT has processed the last data of a scan.
1	DecInProg	For each scan this signal is asserted after the SigSOS (Start of Scan Segment) signal has been output from the core and is de-asserted when the decoding of a scan is complete. It indicates that the core is in the decoding state.
0	JpgInProg	Set when core starts to process input data (JpgIn) and de-asserted when decoding has been completed i.e. when the last pixel of last block of the image is output.

22.5.7 Half-block buffer interface

Since the CDU writes 256 bits (4 x 64 bits) to memory at a time, it requires a double-buffer of 2 x 256 bits at its output, each buffer is a half JPEG block, i.e. 32 bytes stored as 4 x 64-bits. This requires us to be able to stall the JPEG decoder core at its output on a half JPEG block boundary, i.e. after 32 pixels (8 bits per pixel). We provide a mechanism for stalling the JPEG decoder core by gating the clock to the core when *jpg_core_stall* is 1. The half-block buffer interface is responsible for providing a set of double buffered half JPEG blocks to decouple JPEG decoding (read control unit) from writing those JPEG blocks to DRAM (write control unit). Data coming in is 8-bit quantities but data going out is in 64-bit quantities for only a single color plane. Data exits in the same order it enters.

The half-block buffer interface therefore consists of 2 single JPEG half-block buffers and some simple combinatorial logic, as shown in Figure 102.
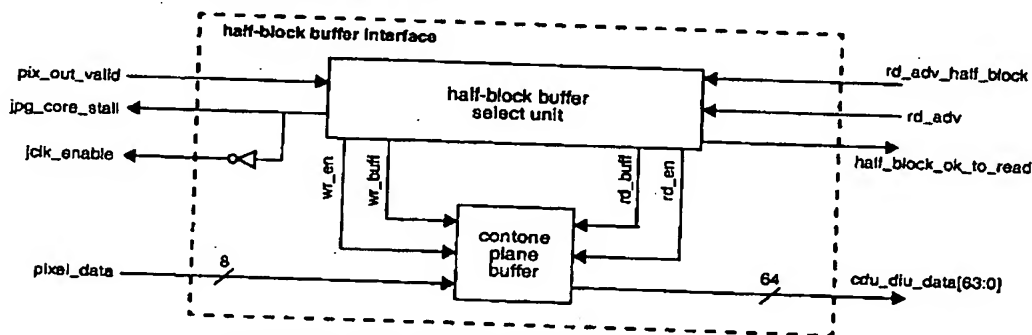

Figure 102. Block diagram of half-block buffer interface

22.5.7.1 Half-block buffer select unit

The half-block buffer select unit provides the mechanism for keeping track of the current read and write buffers, and providing the mechanism such that a buffer cannot be read from until it has been written to. In this case, each buffer is a half JPEG block, i.e. 32 bytes stored as 4 x 64-bits.

The half-block buffer unit keeps a bit for the status of each half-block buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to. The output value *half_block_ok_to_read* equals *buff_avail[rd_buff]*. The output value *jpg_core_stall* equals *buff_avail[wr_buff]*. When *jpg_core_stall* is 1, the clock to the JPEG decoder core is gated off so as to stop the production of pixels. The clock gating is performed in the CPR block under control of the *jclk_enable* output from the CDU. When *jclk_enable* is 1, *jclk* equals *pclk*. When *jclk_enable* is 0, *jclk* is 0 (*jclk_enable* is the inverse of *jpg_core_stall*).

When a *rd_adv_half_block* pulse is received, *buff_avail[rd_buff]* is cleared, and *rd_buff* is inverted. *pixel_count[4:0]* keeps a count of the number of pixels received from the JPEG decoder core. It is incremented whenever *pix_out_valid* is 1 and wraps around when it reaches its maximum value. When *pixel_count[4:0]* is 31, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted. The output *wr_en* equals *pix_out_valid* ANDed with the inverse of jpg_core_stall. The output *rd_en* equals *half_block_ok_to_read* ANDed with *rd_adv*.

22.5.7.2 Contone plane buffer

Each contone plane buffer consists of two half JPEG block buffers as shown in block diagram form in Figure 103.
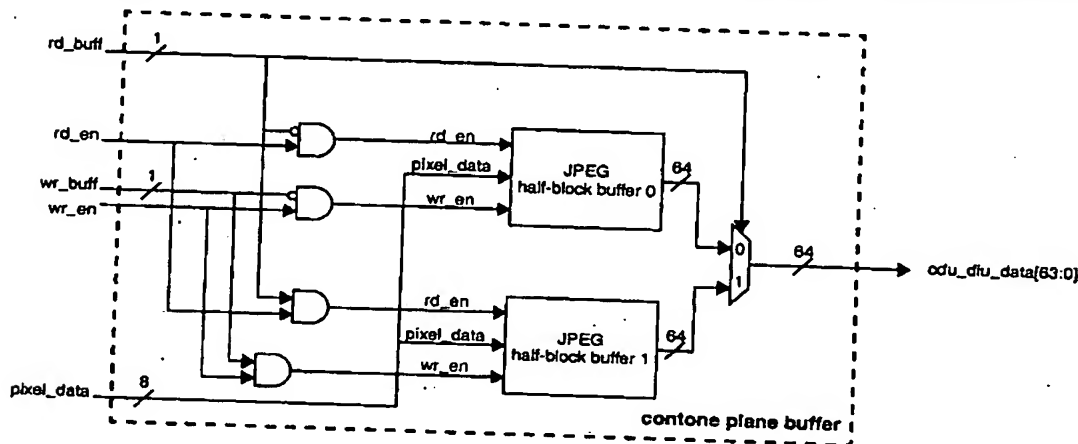
Figure 103. Contone plane buffer interface

Each half JPEG block buffer is implemented by two shift registers and a small amount of combinatorial logic. The first shift register is 7 entries x 8-bit, the second shift register is 4 entry x 64-bit. Data is collected at the first shift register in 8-bit quantities when *wr_en* is 1, and then written to the second shift register in 64 bit quantities. Data is read from the second shift register in 64-bit quantities when *rd_en* is 1.

22.5.8   Write control unit

A line of JPEG blocks in 4 colors, or 8 lines of decompressed contone data, is stored in DRAM with the memory arrangement as shown Figure 104. The arrangement is in order to optimize access for reads by writing the data so that 4 color components are stored together in each 256-bit DRAM word.
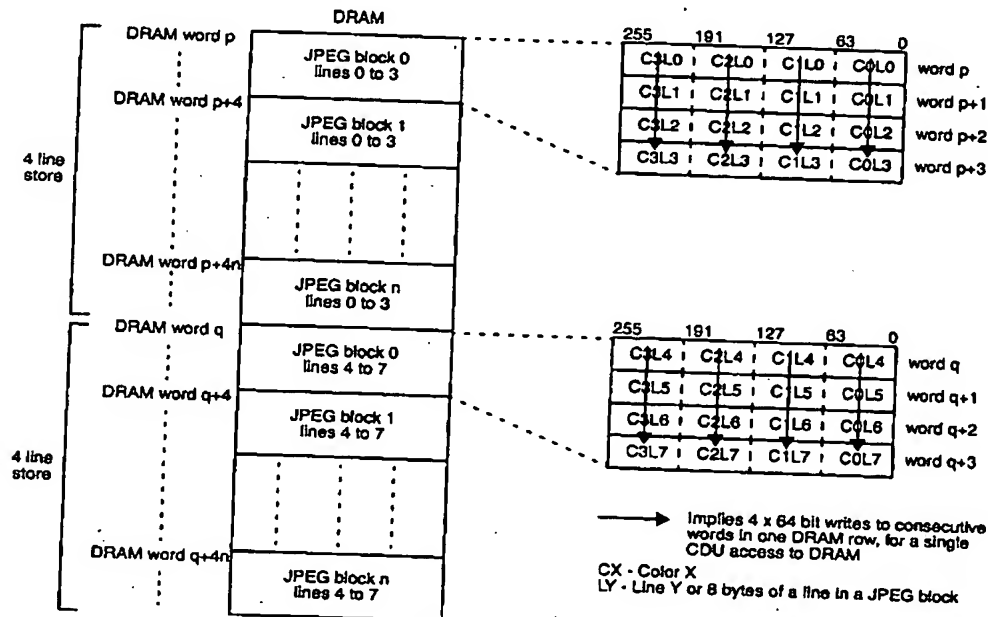

Figure 104. DRAM storage arrangement for a single line of JPEG 8x8 blocks in 4 colors

The CDU writes 8 lines of data in parallel but stores the first 4 lines and second 4 lines separately in DRAM. The write sequence for a single line of JPEG 8x8 blocks in 4 colors, as shown in Figure 104, is as follows below and corresponds to the order in which pixels are output from the JPEG decoder core:

```
block 0, color 0, line 0 in word p bits 63-0, line 1 in word p+1 bits 63-0,
                  line 2 in word p+2 bits 63-0, line 3 in word p+3 bits 63-0,

block 0, color 0, line 4 in word q bits 63-0, line 5 in word q+1 bits 63-0,
                  line 6 in word q+2 bits 63-0, line 7 in word q+3 bits 63-0,

block 0, color 1, line 0 in word p bits 127-64, line 1 in word p+1 bits 127-64,
                  line 2 in word p+2 bits 127-64, line 3 in word p+3 bits 127-64,

block 0, color 1, line 4 in word q bits 127-64, line 5 in word q+1 bits 127-64,
                  line 6 in word q+2 bits 127-64, line 7 in word q+3 bits 127-64,

     repeat for block 0 color 2, block 0 color 3........

block 1, color 0, line 0 in word p+4 bits 63-0, line 1 in word p+5 bits 63-0,

     etc...............................................

block N, color 3, line 4 in word q+4n bits 255-192, line 5 in word q+4n+1 bits 255-192,
                  line 6 in word q+4n+2 bits 255-192, line 7 in word q+4n+3 bit 255-192
```

In SoPEC data is written to DRAM 256 bits at a time. The DIU receives a 64-bit aligned address from the CDU, i.e. the lower 2 bits indicate which 64-bits within a 256-bit location are being written to. With that address the DIU also receives half a JPEG block (4 lines) in a single color, 4 x 64 bits over 4 cycles. All accesses to DRAM must be padded to 256 bits or the bits which should not be written are masked using the individual bit write inputs of the DRAM. When writing decompressed contone data from the CDU, only 64 bits out of the 256-bit access to DRAM are valid, and the remaining bits of the write are masked by the DIU. This means that the decompressed contone data is written to DRAM in 4 back-to-back 64-bit write masked accesses to 4 consecutive 256-bit DRAM locations/words.

Writing of decompressed contone data to DRAM is implemented by the state machine in Figure 105. The CDU writes the decompressed contone data to DRAM half a JPEG block at a time, 4 x 64 bits over 4 cycles. All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *half_block_ok_to_read* and *line_store_ok_to_write* flags to tell it whether to attempt to write a half JPEG block to DRAM. Once the half-block buffer interface contains a half JPEG block, the state machine requests a write access to DRAM by asserting *cdu_diu_wreq* and providing the write address, corresponding to the first 64-bit value to be written, on *cdu_diu_wadr* (only the address the first 64-bit value in each access of 4x64 bits is issued by the CDU. The DIU can generate the addresses for the second, third and fourth 64-bit values). The state machine then waits to receive an acknowledge from the DIU before initiating a read of 4 64-bit values from the half-block buffer interface by asserting *rd_adv* for 4 cycles. The output *cdu_diu_wvalid* is asserted in the cycle after *rd_adv* to indicate to the DIU that valid data is present on the *cdu_diu_data* bus and should be written to the specified address in DRAM. A *rd_adv_half_block* pulse is then sent to the half-block buffer interface to indicate that the current read buffer has been read and should now be available to be written to again. The state machine then returns to the request state.

The pseudocode below shows how the write address is calculated on a per clock cycle basis. Note counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should be cleared and *lwr_halfblock_adr* gets loaded with *buff_start_adr* and *upr_halfblock_adr* gets loaded with *buff_start_adr + max_block + 1*.

```
// assign write address output to DRAM
   cdu_diu_wadr[6:5] = 00          // corresponds to linenumber, only first address is
                                   // issued for each DRAM access. Thus line is always 0.
   cdu_diu_wadr[4:3] = color       // The DIU generates these bits of the address.

   if (half == 1) then
     cdu_diu_wadr[21:7] = upr_halfblock_adr      // for lines 4-7 of JPEG block
   else
     cdu_diu_wadr[21:7] = lwr_halfblock_adr      // for lines 0-3 of JPEG block

// update half, color, block and addresses after each DRAM write access
   if (rd_adv_half_block == 1) then
     if (half == 1) then
       half = 0
       if (color == max_plane) then
         color = 0
         if (block == max_block) then         // end of writing a line of JPEG blocks
           pulse wradv8line
           block = 0

           // update half block address for start of next line of JPEG blocks taking
           // account of address wrapping in circular buffer and 4 line offset
           if (upr_halfblock_adr == buff_end_adr) then
             upr_halfblock_adr = buff_start_adr + max_block + 1
           elsif (upr_halfblock_adr + max_block + 1 == buff_end_adr) then
             upr_halfblock_adr = buff_start_adr
           else
```

```
                          upr_halfblock_adr = upr_halfblock_adr + max_block + 2
                  else
                      block ++
                      upr_halfblock_adr ++              // move to address for lines 4-7 for next block
              else
                  color ++
          else
              half = 1
              if (color == max_plane) then
                  if (block == max_block) then    // end of writing a line of JPEG blocks

                      // update half block address for start of next line of JPEG blocks taking
                      // account of address wrapping in circular buffer and 4 line offset
                      if (lwr_halfblock_adr == buff_end_adr) then
                          lwr_halfblock_adr = buff_start_adr + max_block + 1
                      elsif (lwr_halfblock_adr + max_block + 1 == buff_end_adr) then
                          lwr_halfblock_adr = buff_start_adr
                      else
                          lwr_halfblock_adr = lwr_halfblock_adr + max_block + 2

                  else
                      lwr_halfblock_adr ++              // move to address for lines 0-3 for next block
```
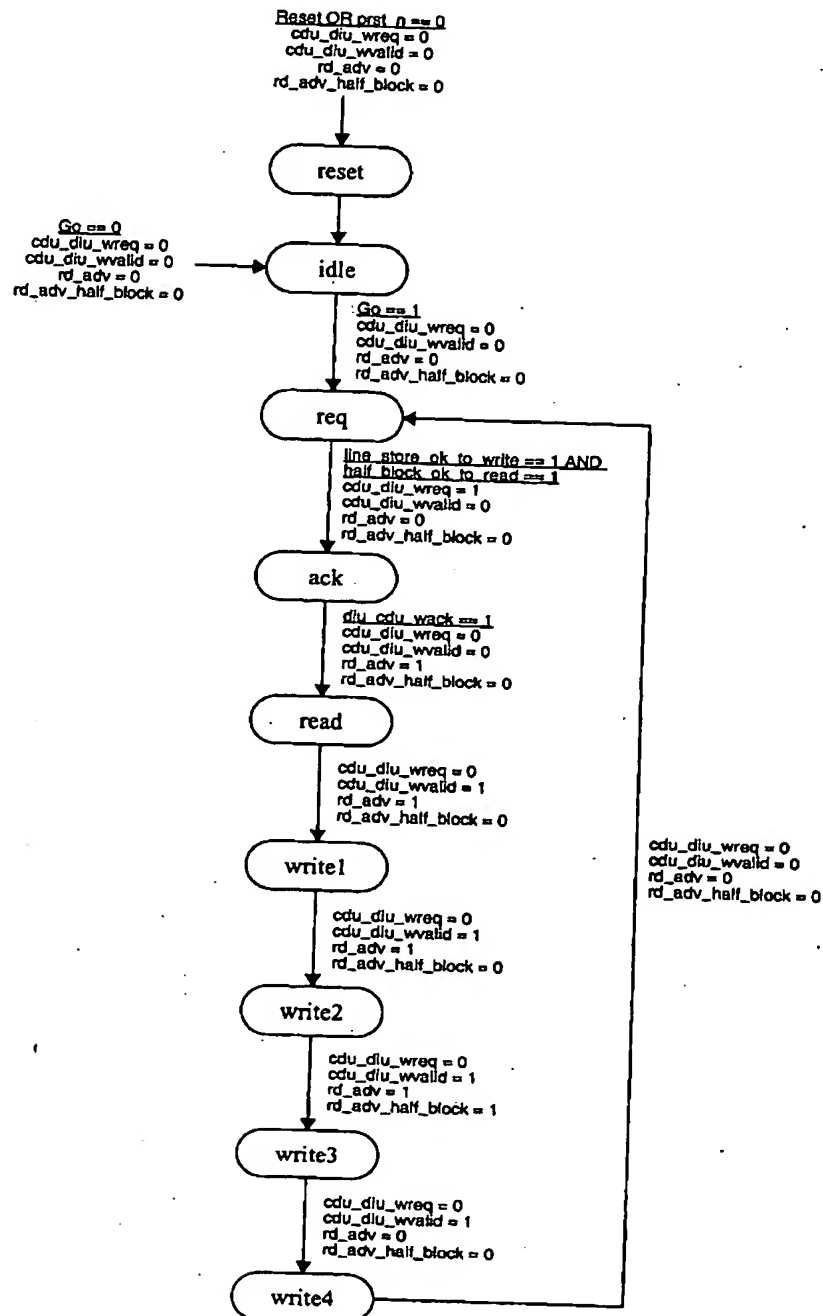
Figure 105. State machine to write decompressed contone data

22.5.9   Contone line store Interface

The contone line store interface is responsible for providing the control over the shared resource in DRAM. The CDU writes 8 lines of data in up to 4 color planes, and the CFU reads them line-at-a-time. The contone line store interface provides the mechanism for keeping track of the number of lines stored in DRAM, and provides signals so that a given line cannot be read from until the complete line has been written.

The CDU writes 8 lines of data in parallel but writes the first 4 lines and second 4 lines to separate areas in DRAM. Thus, when the CFU has read 4 lines from DRAM that area now becomes free for the CDU to write to. Thus the size of the line store in DRAM should be a multiple of 4 lines. The minimum size of the line store interface is 8 lines, providing a single buffer scheme. Typical sizes are 12 lines for a 1.5 buffer scheme while 16 lines provides a double-buffer scheme.

The size of the contone line store is defined by *num_buff_lines*. A count is kept of the number of lines stored in DRAM that are available to be written to. When *Go* transitions from 0 to 1, *num_lines_avail* is set to the value of *num_buff_lines*. The CDU may only begin to write to DRAM as long as there is space available for 8 lines, indicated when the *line_store_ok_to_write* bit is set. When the CDU has finished writing 8 lines, the write control unit sends an *wradv8line* pulse to the contone line store interface and the CFU, and *num_lines_avail* is decremented by 8. The write control unit then waits for *line_store_ok_to_write* to be set again. The CFU is responsible for responding to *wradv8line* pulses appropriately, and sends its own *rdadvline* signal to the CDU's contone line store interface to free up each line as it finishes reading them. *num_lines_avail* is incremented by 1 on receiving a *rdadvline* pulse.

23 Contone FIFO Unit (CFU)

23.1 OVERVIEW

The Contone FIFO Unit (CFU) is responsible for reading the decompressed contone data layer from the circular buffer in DRAM, performing optional color conversion from YCrCb to RGB followed by optional color inversion in up to 4 color planes, and then feeding the data on to the HCU. Scaling of data is performed in the horizontal and vertical directions by the CFU so that the output to the HCU matches the printer resolution. Non-integer scaling is supported in both the horizontal and vertical directions. Typically, the scale factor will be the same in both directions but may be programmed to be different.

23.2 BANDWIDTH REQUIREMENTS

The CFU must read the contone data from DRAM fast enough to match the rate at which the contone data is consumed by the HCU.

Pixels of contone data are replicated a X scale factor (SF) number of times in the X direction and Y scale factor (SF) number of times in the Y direction to convert the final output to 1600 dpi. Replication in the X direction is performed at the output of the CFU on a pixel-by-pixel basis while replication in the Y direction is performed by the CFU reading each line a number of times, according to the Y-scale factor, from DRAM. The HCU generates 1 dot (bi-level in 6 colors) per system clock cycle to achieve a print speed of 1 side per 2 seconds for full bleed A4/Letter printing. The CFU output buffer needs to be supplied with a 4 color contone pixel (32 bits) every SF cycles. With support for 4 colors at 267 ppi the CFU must read data from DRAM at **5.33 bits/cycle**[1].

23.3 COLOR SPACE CONVERSION

The CFU allows the contone data to be passed directly on, which will be the case if the color represented by each color plane in the JPEG image is an available ink. For example, the four colors may be C, M, Y, and K, directly represented by CMYK inks. The four colors may represent gold, metallic green etc. for multi-SoPEC printing with exact colors.

JPEG produces better compression ratios for a given visible quality when luminance and chrominance channels are separated. With CMYK, K can be considered to be luminance, but C, M and Y each contain luminance information and so would need to be compressed with appropriate luminance tables. We therefore provide the means by which CMY can be passed to SoPEC as YCrCb. K does not need color conversion.

When being JPEG compressed, CMY is typically converted to RGB, then to YCrCb and then finally JPEG compressed. At decompression, the YCrCb data is obtained, then color converted to RGB, and finally back to CMY.

The external RIP provides conversion from RGB to YCrCb, specifically to match the actual hardware implementation of the inverse transform within SoPEC, as per CCIR 601-2 [20] except that Y, Cr and Cb are normalized to occupy all 256 levels of an 8-bit binary encoding.

The CFU provides the translation to either RGB or CMY. RGB is included since it is a necessary step to produce CMY, and some printers increase their color gamut by including RGB inks as well as CMYK.

---

1. 32 bits / 6 cycles = 5.33 bits/cycle

Consequently the JPEG stream in the color space convertor is one of:
- 1 color plane, no color space conversion
- 2 color planes, no color space conversion
- 3 color planes, no color space conversion
- 3 color planes YCrCb, conversion to RGB
- 4 color planes, no color space conversion
- 4 color planes YCrCbX, conversion of YCrCb to RGB, no color conversion of X

The YCrCb to RGB conversion is described in [14]. Note that if the data is non-compressed, there is no specific advantage in performing color conversion (although the CDU and CFU do permit it).

23.4 COLOR SPACE INVERSION

In addition to performing optional color conversion the CFU also provides for optional bit-wise inversion in up to 4 color planes. This provides the means by which the conversion to CMY may be finalised, or to may be used to provide planar correlation of the dither matrices.

The RGB to CMY conversion is given by the relationship:
- $C = 255 - R$
- $M = 255 - G$
- $Y = 255 - B$

These relationships require the page RIP to calculate the RGB from CMY as follows:
- $R = 255 - C$
- $G = 255 - M$
- $B = 255 - Y$

23.5 SCALING

Scaling of pixel data is performed in the horizontal and vertical directions by the CFU so that the output to the HCU matches the printer resolution. The CFU supports non-integer scaling with the scale factor represented by a numerator and a denominator. Only scaling up of the pixel data is allowed, i.e. the numerator should be greater than or equal to the denominator. For example, to scale up by a factor of two and a half, the numerator is programmed as 5 and the denominator programmed as 2.

Scaling is implemented using a counter as described in the pseudocode below. An advance pulse is generated to move to the next dot (x-scaling) or line (y-scaling).

```
if (count + denominator - numerator >= 0) then
   count = count + denominator - numerator
   advance = 1
else
   count = count + denominator
   advance = 0
```

23.6 LEAD-IN AND LEAD-OUT CLIPPING

The JPEG algorithm encodes data on a block by block basis, each block consists of 64 8-bit pixels (representing 8 rows each of 8 pixels). If the image is not a multiple of 8 pixels in X and Y then padding must be present. This padding (extra pixels) will be present after decoding of the JPEG bytestream.

Extra padded lines in the Y direction (which may get scaled up in the CFU) will be ignored in the HCU through the setting of the *BottomMargin* register.

Extra padded pixels in the X direction must also be removed so that the contone layer is clipped to the target page as necessary.

In the case of a multi-SoPEC system, 2 SoPECs may be responsible for printing the same side of a page, e.g. SoPEC #1 controls printing of the left side of the page and SoPEC #2 controls printing of the right side of the page and shown in Figure 106. The division of the contone layer between the 2 SoPECs may not fall on a 8 pixel (JPEG block) boundary. The JPEG block on the boundary of the 2 SoPECs (JPEG block n below) will be the last JPEG block in the line printed by SoPEC #1 and the first JPEG block in the line printed by SoPEC #2. Pixels in this JPEG block not destined for SoPEC #1 are ignored by appropriately setting the *LeadOutClipNum*. Pixels in this JPEG block not destined for SoPEC #2 must be ignored at the beginning of each line. The number of pixels to be ignored at the start of each line is specified by the *LeadInClipNum* register.

It may also be the case that the CDU writes out more JPEG blocks than is required to be read by the CFU, as shown for SoPEC #2 below. In this case the value of the *MaxBlock* register in the CDU is set to correspond to JPEG block m but the value for the *MaxBlock* register in the CFU is set to correspond to JPEG block $m-1$. Thus JPEG block m is not read in by the CFU.
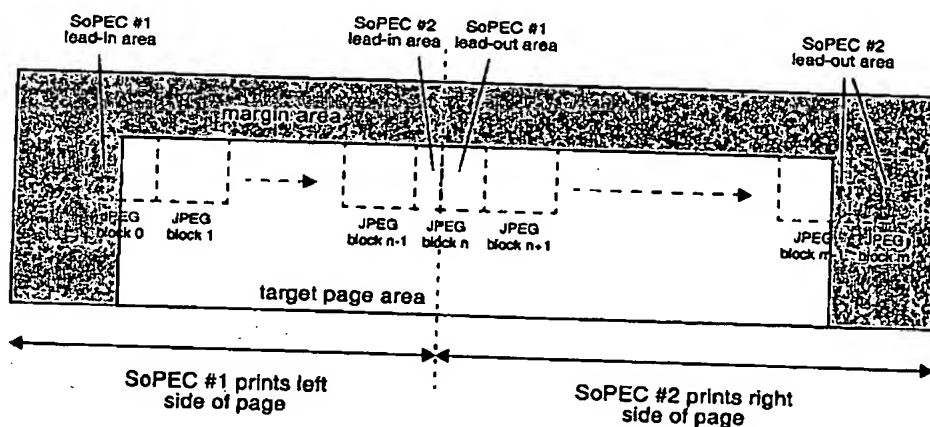
Figure 106. Lead-in and lead-out clipping of contone data in multi-SoPEC environment

Additional clipping on contone pixels is required when they are scaled up to the printer's resolution. The scaling of the first valid pixel in the line is controlled by setting the *XstartCount* register. The *HcuLine-Length* register defines the size of the target page for the contone layer at the printer's resolution and controls the scaling of the last valid pixel in a line sent to the HCU.

23.7 IMPLEMENTATION

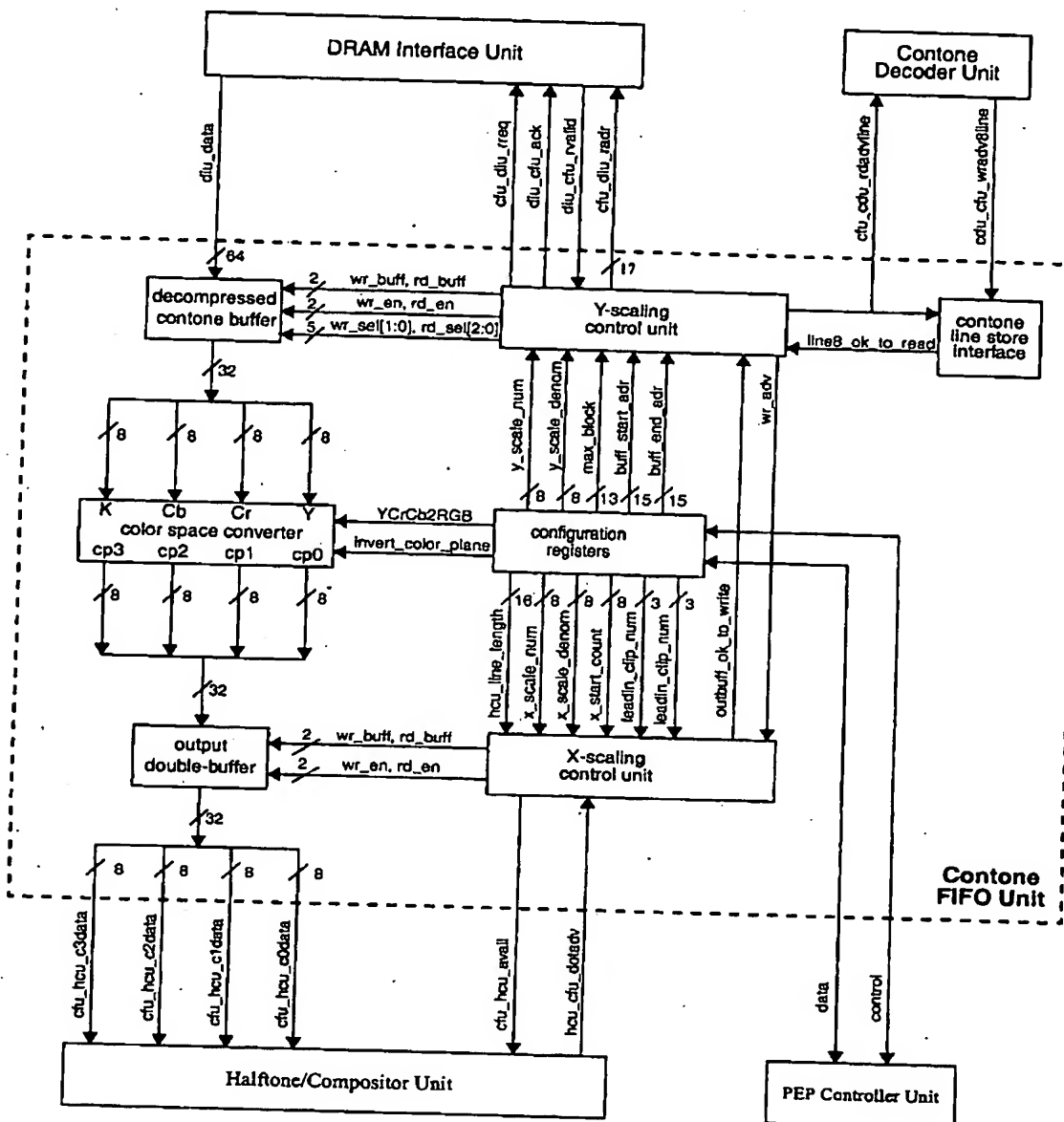Figure 107 shows a block diagram of the CFU.

Figure 107. Block diagram of CFU

23.7.1 Definitions of I/O

Table 101. CFU port list and description

Port Name	Pins	I/O	Description
Clocks and reset			
pclk	1	In	System clock
prst_n	1	In	System reset, synchronous active low.
PCU Interface			
pcu_cfu_sel	1	In	Block select from the PCU. When *pcu_cfu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[6:2]	4	In	PCU address bus. Only 5 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
cfu_pcu_rdy	1	Out	Ready signal to the PCU. When *cfu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *cfu_pcu_data* is valid.
cfu_pcu_data[31:0]	32	Out	Read data bus to the PCU.
DIU Interface			
cfu_diu_rreq	1	Out	CFU read request, active high. A read request must be accompanied by a valid read address.
diu_cfu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *cfu_diu_radr*.
cfu_diu_radr[21:5]	17	Out	CFU read address. 17 bits wide (256-bit aligned word).
diu_cfu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DRAM.
CDU Interface			
cdu_cfu_wradv8line	1	In	Write 8line pulse, active high. Indicates that the CDU has finished writing to 8 lines of decompressed contone data to the circular buffer in DRAM and the data is available to be read by the CFU.
cfu_cdu_rdadvline	1	Out	Read line pulse, active high. Indicates that the CFU has finished reading a line of decompressed contone data to the circular buffer in DRAM and that line of the buffer is now free.
HCU Interface			
hcu_cfu_advdot	1	In	Informs the CFU that the HCU has captured the pixel data on cfu_hcu_c[0-3]data lines and the CFU can now place the next pixel on the data lines.
cfu_hcu_avail	1	Out	Indicates valid data present on cfu_hcu_c[0-3]data lines.
cfu_hcu_c0data[7:0]	8	Out	Pixel of data in contone plane 0.
cfu_hcu_c1data[7:0]	8	Out	Pixel of data in contone plane 1.
cfu_hcu_c2data[7:0]	8	Out	Pixel of data in contone plane 2.
cfu_hcu_c3data[7:0]	8	Out	Pixel of data in contone plane 3.

23.7.2   Configuration registers

The configuration registers in the CFU are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for the description of the protocol and timing diagrams for reading and writing registers in the CFU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the CFU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *cfu_pcu_data*. The configuration registers of the CFU are listed in Table 102:

Table 102. CFU registers

Address (CFU base +)	Register Name	#bits	Value on Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the CFU.
0x04	Go	1	0x0	Writing 1 to this register starts the CFU. Writing 0 to this register halts the CFU. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The CFU must be started before the CDU is started. This register can be read to determine if the CFU is running (1 - running, 0 - stopped).
Setup registers				
0x10	MaxBlock	13	0x000	Number of JPEG MCUs (or JPEG block equivalents, i.e. 8x8 bytes) in a line - 1.
0x14	BuffStartAdr	15	0x0000	Points to the start of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary. A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x18	BuffEndAdr	15	0x0000	Points to the end of the decompressed contone circular buffer in DRAM, aligned to a half JPEG block boundary (address is inclusive). A half JPEG block consists of 4 words of 256-bits, enough to hold 32 contone pixels in 4 colors, i.e. half a JPEG block.
0x1C	4LineOffset	13	0x0000	Defines the offset between the start of one 4 line store to the start of the next 4 line store. In Figure 108 on page 294, if *BufStartAdr* corresponds to line 0 block 0 then *BuffStartAdr + 4LineOffset* corresponds to line 4 block 0. This register is required in addition to *MaxBlock* as the number of JPEG blocks in a line required by the CFU may be different from the number of JPEG blocks in a line written by the CDU.
0x20	YCrCb2RGB	1	0x0	Set this bit to enable conversion from YCrCb to RGB. Should not be changed between bands.

Table 102. CFU registers

Address (CFU_base+)	Register Name	No. of bits	Value on Reset	Description

0x24	InvertColorPlane	4	0x0	Set these bits to perform bit-wise inversion on a per color plane basis.
				bit0 - 1 invert color plane 0
				    - 0 do not convert
				bit1 - 1 invert color plane 1
				    - 0 do not convert
				bit2 - 1 invert color plane 2
				    - 0 do not convert
				bit3 - 1 invert color plane 3
				Should not be changed between bands.

0x28	HcuLineLength	16	0x0000	Number of contone pixels - 1 in a line (after scaling). Equals the number of *hcu_cfu_dotadv* pulses - 1 received from the HCU for each line of contone data.
0x2C	LeadInClipNum	3	0x0	Number of contone pixels to be ignored at the start of a line (from JPEG block 0 in a line). They are not passed to the output buffer to be scaled in the X direction.
0x30	LeadOutClipNum	3	0x0	Number of contone pixels to be ignored at the end of a line (from JPEG block *MaxBlock* in a line). They are not passed to the output buffer to be scaled in the X direction.
0x34	XstartCount	8	0x00	Value to be loaded at the start of every line into the counter used for scaling in the X direction. Used to control the scaling of the first pixel in a line to be sent to the HCU. This value will typically be zero, except in the case where a number of dots are clipped on the lead in to a line.
0x38	XscaleNum	8	0x01	Numerator of contone scale factor in X direction.
0x3C	XscaleDenom	8	0x01	Denominator of contone scale factor in X direction.
0x40	YscaleNum	8	0x01	Numerator of contone scale factor in Y direction.
0x44	YscaleDenom	8	0x01	Denominator of contone scale factor in Y direction.

23.7.3 Storage of decompressed contone data in DRAM

The CFU reads decompressed contone data from DRAM in single 256-bit accesses. JPEG blocks of decompressed contone data are stored in DRAM with the memory arrangement as shown The arrangement is in order to optimize access for reads by writing the data so that 4 color components are stored together in each 256-bit DRAM word. The means that the CFU reads 64-bits in 4 colors from a single line in each 256-bit DRAM access.
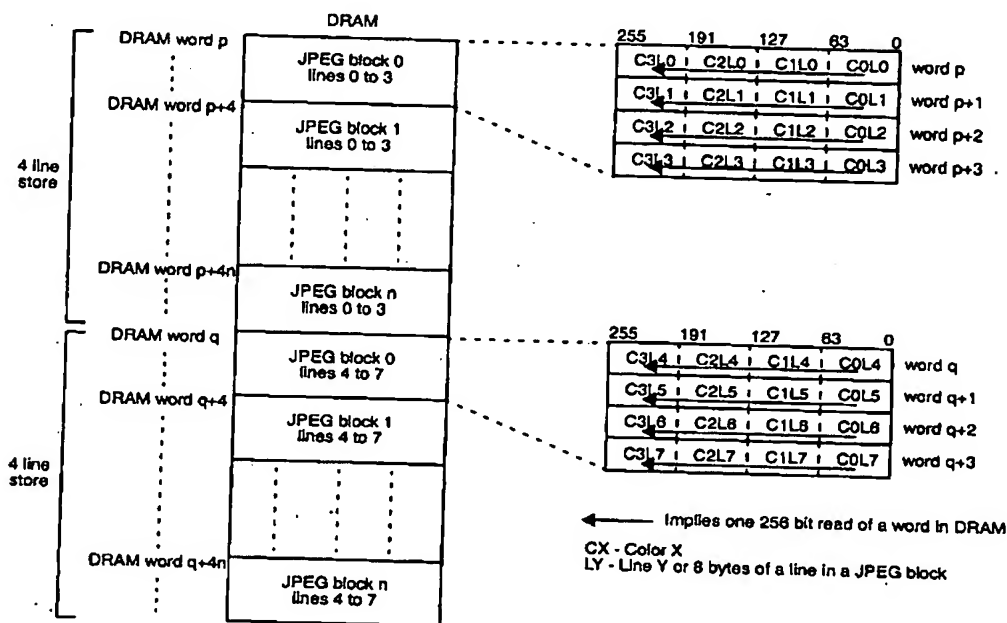
Figure 108. DRAM storage arrangement for a single line of JPEG blocks in 4 colors

The CFU reads data line at a time in 4 colors from DRAM. The read sequence, as shown in Figure 108, is as follows:

```
line 0, block 0 in word p of DRAM
line 0, block 1 in word p+4 of DRAM
..................................................
line 0, block n in word p+4n of DRAM
(repeat to read line a number of times according to scale factor)

line 1, block 0 in word p+1 of DRAM
line 1, block 1 in word p+5 of DRAM
etc....................................
```

The CFU reads a complete line in up to 4 colors a Y scale factor number of times from DRAM before it moves on to read the next. When the CFU has finished reading 4 lines of contone data that 4 line store becomes available for the CDU to write to.

23.7.4 Decompressed contone buffer

Since the CFU reads 256 bits (4 colors x 64 bits) from memory at a time, it requires storage of 2 x 256 bits at its input. The CFU receives the data from the DIU over 4 clock cycles (64-bits of a single color per cycle). It is implemented as 4 buffers. Each buffer conceptually is a 64-bit input and 8-bit output buffer to account for the 64-bit data transfers from the DIU, and the 8-bit output per color plane to the color space converter. In reality, each buffer is actually implemented as a double-buffer of 2 x 64-bits wide.

On the DRAM side, *wr_buff* indicates the current buffer within each double-buffer that writes are to occur to. *wr_sel* selects which double-buffer to write the 64 bits of data to when *wr_en* is asserted.

On the color space converter side, *rd_buff* indicates the current buffer within each double-buffer that reads are to occur from. When *rd_en* is asserted a byte is read from each of the double-buffers in parallel. *rd_sel* is used to select a byte from the 64 bits (1st byte corresponds to bits 7-0, second byte to bits 15-8 etc.).

23.7.5 Y-scaling control unit

The Y-scaling control unit is responsible for reading the decompressed contone data and passing it to the color space converter via the decompressed contone buffer. The decompressed contone data is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 208. Read accesses to DRAM are implemented by means of the state machine described in Figure 109.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is set, the state machine relies on the *line8_ok_to_read* and *buff_ok_to_write* flags to tell it whether to attempt to read a line of compressed contone data from DRAM. When *line8_ok_to_read* is 0 the state machine does nothing. When *line8_ok_to_read* is 1 the state machine continues to load data into the decompressed contone buffer up to 256-bits at a time while there is space available in the buffer.

A bit is kept for the status of each 64-bit buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to.

buff_ok_to_write equals ~*buff_avail[wr_buff]*. When a *wr_adv_buff* pulse is received, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted. Whenever *diu_cfu_rvalid* is asserted, *wr_en* is asserted to write the 64-bits of data from DRAM to the buffer selected by *wr_sel* and *wr_buff*.

buff_ok_to_read equals *buff_avail[rd_buff]*. If there is data available in the buffer and the output double-buffer has space available (*outbuff_ok_to_write* equals 1) then data is read from the buffer by asserting *rd_en* and *rd_sel* gets incremented to point to the next value. *wr_adv* is asserted in the following cycle to write the data to the output double-buffer of the CFU. When finished reading the buffer, *rd_sel* equals b111 and *rd_en* is asserted, *buff_avail[rd_buff]* is set, and *rd_buff* is inverted.

Each line is read a number of times from DRAM, according to the Y-scale factor, before the CFU moves on to start reading the next line of decompressed contone data. Scaling to the printhead resolution in the Y direction is thus performed.

The pseudocode below shows how the read address from DRAM is calculated on a per clock cycle basis. Note all counters and flags should be cleared after reset or when *Go* is cleared. When a 1 is written to *Go*, both *curr_halfblock* and *line_start_halfblock* get loaded with *buff_start_adr*, and *y_scale_count* gets loaded with *y_scale_denom*. Scaling in the Y direction is implemented by line replication by re-reading lines from DRAM. The algorithm for non-integer scaling is described in the pseudocode below.

```
// assign read address output to DRAM
   cdu_diu_wadr[21:7] = curr_halfblock
   cdu_diu_wadr[6:5]  = line[1:0]

// update block, line, y_scale_count and addresses after each DRAM read access
   if (wr_adv_buff == 1)
       if (block == max_block) then    // end of reading a line of contone in up to 4 colors
           block = 0

           // check whether to advance to next line of contone data in DRAM
           if (y_scale_count + y_scale_denom - y_scale_num >= 0) then
               y_scale_count = y_scale_count + y_scale_denom - y_scale_num
               pulse RdAdvline
               if (line == 3) then     // end of reading 4 line store of contone data
```

```
            line = 0

            // update half block address for start of next line taking account of
            // address wrapping in circular buffer and 4 line offset
            if (curr_halfblock == buff_end_adr) then
               curr_halfblock = buff_start_adr
               line_start_adr = buff_start_adr
            elsif ((line_start_adr + 4line_offset) == buff_end_adr)) then
               curr_halfblock = buff_start_adr
               line_start_adr = buff_start_adr
            else
               curr_halfblock = line_start_adr + 4line_offset
               line_start_adr = line_start_adr + 4line_offset

         else
            line ++
            curr_halfblock = line_start_adr
      else
         // re-read current line from DRAM
         y_scale_count = y_scale_count + y_scale_denom
         curr_halfblock = line_start_adr
   else
      block ++
      curr_halfblock ++
```
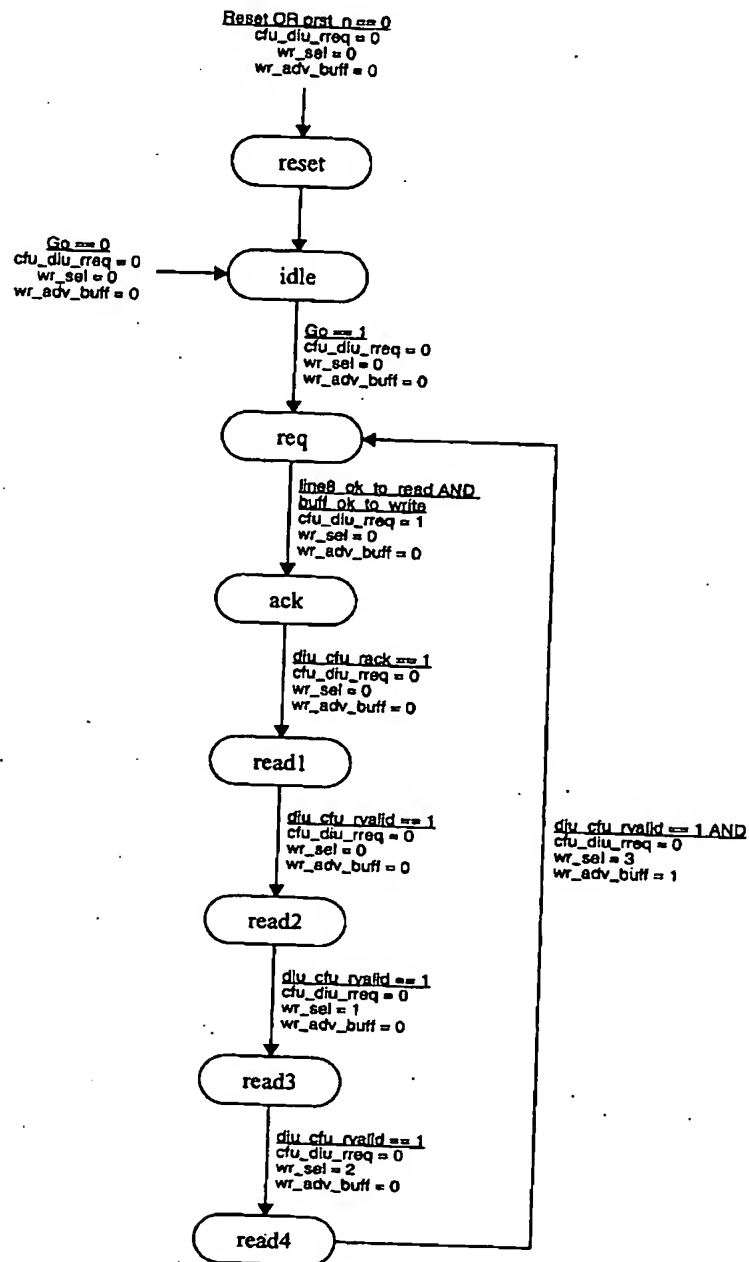
Figure 109. State machine to read decompressed contone data from DRAM

23.7.6 Contone line store interface

The contone line store interface is responsible for providing the control over the shared resource in DRAM. The CDU writes 8 lines of data in up to 4 color planes, and the CFU reads them line-at-a-time. The contone line store interface provides the mechanism for keeping track of the number of lines stored in DRAM, and provides signals so that a given line cannot be read from until the complete line has been written.

A count is kept of the number of lines that have been written to DRAM by the CDU and are available to be read by the CFU. At start-up, *buff_lines_avail* is set to the 0. The CFU may only begin to read from DRAM when the CDU has written 8 complete lines of contone data. When the CDU has finished writing 8 lines, it sends an *cdu_cfu_wradv8line* pulse to the CFU, and *buff_lines_avail* is incremented by 8. The CFU may continue reading from DRAM as long as *buff_lines_avail* is greater than 0. *line8_ok_to_read* is set while *buff_lines_avail* is greater than 0. When it has completely finished reading a line of contone data from DRAM, the Y-scaling control unit sends a *RdAdvLine* signal to contone line store interface and to the CDU to free up the line in the buffer in DRAM. *buff_lines_avail* is decremented by 1 on receiving a *RdAdvline* pulse.

23.7.7 Color Space Converter (CSC)

The color space converter consists of 2 stages: optional color conversion from YCrCb to RGB followed by optional bit-wise inversion in up to 4 color planes.

The convert YCrCb to RGB block takes 3 8-bit inputs defined as Y, Cr, and Cb and outputs either the same data YCrCb or RGB. The YCrCb2RGB parameter is set to enable the conversion step from YCrCb to RGB. If YCrCb2RGB equals 0, the conversion does not take place, and the input pixels are passed to the second stage. The 4th color plane, if present, bypasses the convert YCrCb to RGB block. Note that the latency of the convert YCrCb to RGB block is 1 cycle. This latency should be equalized for the 4th color plane as it bypasses the block.

The second stage involves optional bit-wise inversion on a per color plane basis under the control of *invert_color_plane*. For example if the input is YCrCbK, then *YCrCb2RGB* can be set to 1 to convert YCrCb to RGB, and *invert_color_plane* can be set to 0111 to then convert the RGB to CMY, leaving K unchanged.

If *YCrCb2RGB* equals 0 and *invert_color_plane* equals 0000, no color conversion or color inversion will take place, so the output pixels will be the same as the input pixels.

Figure 110 shows a block diagram of the color space converter.
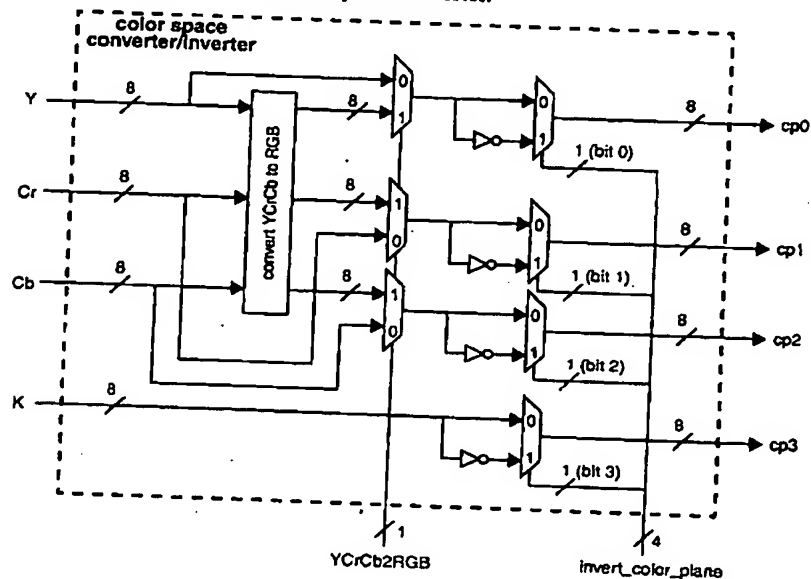
Figure 110. Block diagram of color space converter

The convert YCrCb to RGB block is an implementation of [14]. Although only 10 bits of coefficients are used (1 sign bit, 1 integer bit, 8 fractional bits), full internal accuracy is maintained with 18 bits. The conversion is implemented as follows:

- $R* = Y + (359/256)(Cr-128)$
- $G* = Y - (183/256)(Cr-128) - (88/256)(Cb-128)$
- $B* = Y + (454/256)(Cb-128)$

$R*$, $G*$ and $B*$ are rounded to the nearest integer and saturated to the range 0-255 to give R, G and B. Note that, while a *Reset* results in all-zero output, a zero input gives output RGB = [0[1], 136[2], 0[3]].

23.7.8 X-scaling control unit

The CFU has a 2 x 32-bit double-buffer at its output between the color space converter and the HCU. The X-scaling control unit performs the scaling of the contone data to the printers output resolution, provides the mechanism for keeping track of the current read and write buffers, and ensures that a buffer cannot be read from until it has been written to.

A bit is kept for the status of each 32-bit buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to.

The output value *outbuff_ok_to_write* equals ~*buff_avail[wr_buff]*. Contone pixels are counted as they are received from the Y-scaling control unit, i.e. when *wr_adv* is 1. Pixels in the lead-in and lead-out areas are

---

1. -179 is saturated to 0

2. 135.5, with rounding becomes 136.

3. -227 is saturated to 0

ignored, i.e. they are not written to the output buffer. Lead-in and lead-out clipping of pixels is implemented by the following pseudocode that generates the *wr_en* pulse for the output buffer.

```
if (wradv == 1) then
    if (pixel_count == {max_block,b111}) then
        pixel_count = 0
    else
        pixel_count ++
    if ((pixel_count < leadin_clip_num)
            OR (pixel_count > ({max_block,b111} - leadout_clip_num))) then
        wr_en = 0
    else
        wr_en = 1
```

When a *wr_en* pulse is sent to the output double-buffer, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted.

The output *cfu_hcu_avail* equals *buff_avail[rd_buff]*. When *cfu_hcu_avail* equals 1, this indicates to the HCU that data is available to be read from the CFU. The HCU responds by asserting *hcu_cfu_advdot* to indicate that the HCU has captured the pixel data on *cfu_hcu_c[0-3]data* lines and the CFU can now place the next pixel on the data lines.

The input pixels from the CSC may be scaled a non-integer number of times in the X direction to produce the output pixels for the HCU at the printhead resolution. Scaling is implemented by pixel replication. The algorithm for non-integer scaling is described in the pseudocode below. Note, *x_scale_count* should be loaded with *x_start_count* after reset and at the end of each line. This controls the amount by which the first pixel is scaled by. *hcu_line_length* and *hcu_cfu_dotadv* control the amount by which the last pixel in a line that is sent to the HCU is scaled by.

```
if (hcu_cfu_dotadv == 1) then
    if (x_scale_count + x_scale_denom - x_scale_num >= 0) then
        x_scale_count = x_scale_count + x_scale_denom - x_scale_num
        rd_en = 1
    else
        x_scale_count = x_scale_count + x_scale_denom
        rd_en = 0
else
    x_scale_count = x_scale_count
    rd_en = 0
```

When a *rd_en* pulse is received, *buff_avail[rd_buff]* is cleared, and *rd_buff* is inverted.

A 16-bit counter, *dot_adv_count*, is used to keep a count of the number of *hcu_cfu_dotadv* pulses received from the HCU. If the value of *dot_adv_count* equals *hcu_line_length* and a *hcu_cfu_dotadv* pulse is received, then a *rd_en* pulse is genrated to present the next dot at the output of the CFU, *dot_adv_count* is reset to 0 and *x_scale_count* is loaded with *x_start_count*.

24 Lossless Bi-level Decoder (LBD)

24.1 OVERVIEW

The Lossless Bi-level Decoder (LBD) is responsible for decompressing a single plane of bi-level data. In SoPEC bi-level data is limited to a single spot color (typically black for text and line graphics).

The input to the LBD is a single plane of bi-level data, read as a bitstream from DRAM. The LBD is programmed with the start address of the compressed data, the length of the output (decompressed) line, and the number of lines to decompress. Although the requirement for SoPEC is to be able to print text at 10:1 compression, the LBD can cope with any compression ratio if the requested DRAM access is available. A pass-through mode is provided for 1:1 compression. Ten-point plain text compresses with a ratio of about 50:1. Lossless bi-level compression across an average page is about 20:1 with 10:1 possible for pages which compress poorly.

The output of the LBD is a single plane of decompressed bi-level data. The decompressed bi-level data is output to the SFU (Spot FIFO Unit), and in turn becomes an input to the HCU (Halftoner/Compositor unit) for the next stage in the printing pipeline. The LBD also outputs a *lbd_finishedband* control flag that is used by the PCU and is available as an interrupt to the CPU.
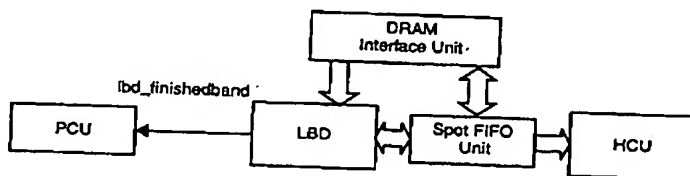
Figure 111. High level block diagram of LBD in context

24.2 MAIN FEATURES OF LBD

Figure 112 shows a schematic outline of the LBD and SFU.

The LBD is required to support compressed images of up to 800 dpi. If possible we would like to support bi-level images of up to 1600 dpi. The line buffers must therefore be long enough to store a complete line at 1600 dpi.

The PEC1 LBD is required to output 2 dots/cycle to the HCU. This throughput capability is retained for SoPEC to minimise changes to the block, although in SoPEC the HCU will only read 1 dot/cycle. The PEC1 LDB outputs 16 bits in parallel to the PEC1 spot buffer. This is also retained for SoPEC. Therefore the LBD in SoPEC can run much faster than is required. This is useful for allowing stalls, e.g. due to band processing latency, to be absorbed.

The LBD has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a runlength code, followed by pass through.

The LBD outputs decompressed bi-level data to the *NextLineFIFO* in the Spot FIFO Unit (SFU). This stores the decompressed lines in DRAM, with a typical minimum of 2 lines stored in DRAM, nominally 3 lines up to a programmable number of lines. The SFU's *NextLineFIFO* can fill while the SFU waits for write access to DRAM. Therefore the LBD must be able to support stalling at its output during a line.

The LBD uses the previous line in the decoding process. This is provided by the SFU via it's *PrevLine-FIFO*. Decoding can stall in the LBD while this FIFO waits to be filled from DRAM.

A signal *sfu_ldb_rdy* indicates that both the SFU's *NextLineFIFO* and *PrevLineFIFO* are available for writing and reading, respectively.

A configuration register in the LBD controls whether the first line being decoded at the start of a band uses the previous line read from the SFU or uses an all 0's line instead.

The line length stored in DRAM must be programmable to multiples of 16 bits, as the LBD output bus is 16 bits. An A4 line of 13824 dots requires 1.7Kbytes of storage. An A3 line of 19488 dots requires 2.4 Kbytes of storage.

The compressed spot data can be read at a rate of 1 bit/cycle for pass through mode 1:1 compression.

The LBD finished band signal is exported to the PCU and is additionally available to the CPU as an interrupt.
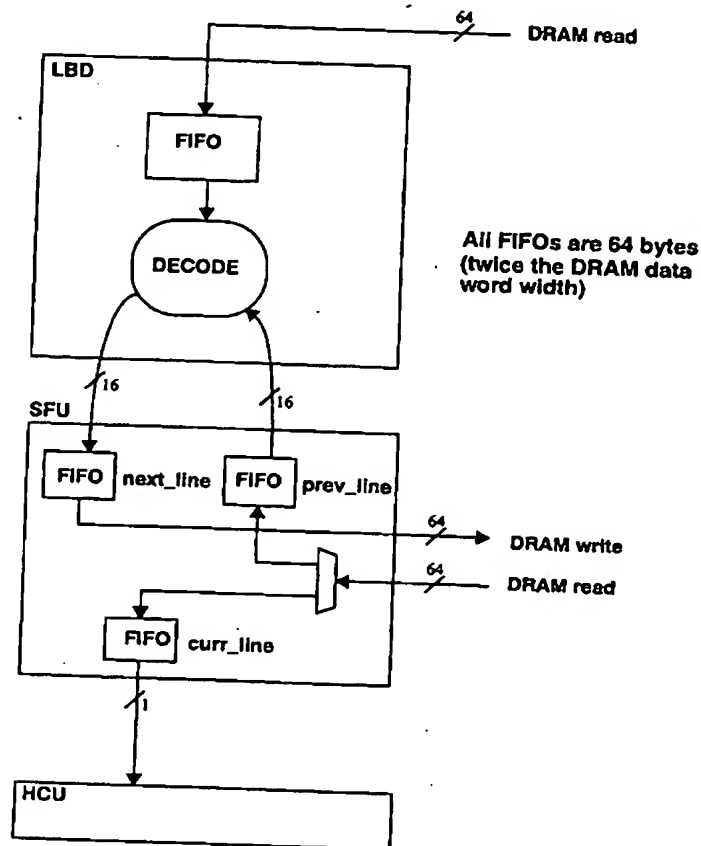
Figure 112. Schematic outline of the LBD and the SFU

24.2.1 Bi-level Decoding in the LBD

The black bi-level layer is losslessly compressed using Silverbrook Modified Group 4 (SMG4) compression which is a version of Group 4 Facsimile compression [18] without Huffman and with simplified run length encodings. The encoding are listed in Table 103 and Table 104

Table 103. Bi-Level group 4 facsimile style compression encodings

	Encoding	Description
same as Group 4 Facsimile	1000	Pass Command: a0 ← b2, skip next two edges
	1	Vertical(0): a0 ← b1, color = !color
	110	Vertical(1): a0 ← b1 + 1, color = !color
	010	Vertical(-1): a0 ← b1 - 1, color = !color
	110000	Vertical(2): a0 ← b1 + 2, color = !color
	010000	Vertical(-2): a0 ← b1 - 2, color = !color
Unique to this implementation	100000	Vertical(3): a0 ← b1 + 3, color = !color
	000000	Vertical(-3): a0 ← b1 - 3, color = !color
	<RL><RL>100	Horizontal: a0 ← a0 + <RL> + <RL>

SMG4 has a pass through mode to cope with local negative compression. Pass through mode is activated by a special run-length code. Pass through mode continues to either end of line or for a pre-programmed number of bits, whichever is shorter. The special run-length code is always executed as a run-length code, followed by pass through. The pass through escape code is a medium length run-length with a run of less than or equal to 31.

Table 104. Run length (RL) encodings

	Encoding	Description
Unique to this implementation	RRRRR1	Short Black Runlength (5 bits)
	RRRRR1	Short White Runlength (5 bits)
	RRRRRRRRRR10	Medium Black Runlength (10 bits)
	RRRRRRRR10	Medium White Runlength (8 bits)
	RRRRRRRRRR10	Medium Black Runlength with RRRRRRRRRR <= 31, Enter pass through
	RRRRRRRR10	Medium White Runlength with RRRRRRRR <= 31, Enter pass through
	RRRRRRRRRRRRRRR00	Long Black Runlength (15 bits)
	RRRRRRRRRRRRRRR00	Long White Runlength (15 bits)

Since the compression is a bitstream, the encodings are read right (least significant bit) to left (most significant bit). The run lengths given as RRRRR in Table 104 are read in the same way (least significant bit at the right to most significant bit at the left).

There is an additional enhancement to the G4 fax algorithm, it relates to pass through mode. It is possible for data to compress negatively using the G4 fax algorithm. On occasions like this it would be easier to pass the data to the LBD as un-compressed data. Pass through mode is a new feature that was not implemented in the PEC1 version of the LBD. When the LBD is in pass through mode the least significant bit of the data stream is an un-compressed bit. This bit is used to construct the current line.

To enter pass through mode the LBD takes advantage of the way run lengths can be written. Usually if one of the runlength pair is less than or equal to 31 it should be encoded as a short runlength. However under the coding scheme of Table 104 it is still legal to write it as a medium or long runlength. The LBD has been designed so that if a short runlength value is detected in a medium runlength then once the horizontal command containing this runlength is decoded completely this will tell the LBD to enter pass through mode and the bits following the runlength is un-compressed data. The number of bits to pass through is either a programmed number of bits or the end of the line which ever comes first. Once the pass through mode is completed the current color is the same as the color of the last bit of the passed through data.

24.2.2   DRAM Access Requirements

The compressed page store for contone, bi-level and raw tag data is 2 Mbytes. The LBD will access the compressed page store in single 256-bit DRAM reads. The LBD will need a 256-bit double buffer in its interface to the DIU. The LBD's DIU bandwidth requirements are summarized in Table 105

Table 105. DRAM bandwidth requirements

Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth (bits/cycle)	Average Bandwidth (bits/cycle)
Read	256[1] (1:1 compression)	1 (1:1 compression)	0.1 (10:1 compression)

1: At 1:1 compression the LBD requires 1 bit/cycle or 256 bits every 256 cycles.

24.3   IMPLEMENTATION

24.3.1   Definitions of IO

Table 106. LBD Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
Bandstore signals			
cdu_endofbandstore[21:5]	17	In	Address of the end of the current band of data. 256-bit word aligned DRAM address.
cdu_startofbandstore[21:5]	17	In	Address of the start of the current band of data. 256-bit word aligned DRAM address.
lbd_finishedband	1	Out	LBD finished band signal to PCU and Interrupt Controller.
DIU Interface signals			
lbd_diu_rreq	1	Out	LBD requests DRAM read. A read request must be accompanied by a valid read address.
lbd_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_lbd_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *lbd_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to SoPEC Units. First 64-bits is bits 63:0 of 256 bit word. Second 64-bits is bits 127:64 of 256 bit word. Third 64-bits is bits 191:128 of 256 bit word. Fourth 64-bits is bits 255:192 of 256 bit word.
diu_lbd_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus
PCU interface data and control signals			
pcu_addr[5:2]	4	In	PCU address bus. Only 4 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
lbd_pcu_datain[31:0]	32	Out	Read data bus from the LBD to the PCU.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_lbd_sel	1	In	Block select from the PCU. When *pcu_lbd_sel* is high both *pcu_addr* and *pcu_dataout* are valid.
lbd_pcu_rdy	1	Out	Ready signal to the PCU. When *lbd_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *lbd_pcu_datain* is valid.
SFU interface data and control signals			
sfu_lbd_rdy	1	In	Ready signal indicating SFU has previous line data available for reading and is also ready to be written to.
lbd_sfu_advline	1	Out	Advance line signal to previous and next line buffers
lbd_sfu_pladvword	1	Out	Advance word signal for previous line buffer.

Table 106. LBD Port List

Port Name	Pins	I/O	Description
sfu_lbd_pidata[15:0]	16	In	Data from the previous line buffer.
lbd_sfu_wdata[15:0]	16	Out	Write data for next line buffer.
lbd_sfu_wdatavalid	1	Out	Write data valid signal for next line buffer data.

24.3.2   Configuration Registers

Table 107. LBD Configuration Registers

Address (LBD base +)	Register Name	# Bits	Value on Reset	description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the LBD. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0x0	Writing 1 to this register starts the LBD. Writing 0 to this register halts the LBD. The *Go* register is reset to 0 by the LBD when it finishes processing a band. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The LBD should only be started after the SFU is started. This register can be read to determine if the LBD is running (1 - running, 0 - stopped).
Setup registers (constant for during processing the page)				
0x08	LineLength	16	0x0000	Width of expanded bi-level line (in dots) (must be a multiple of 16 bits).
0x0C	PassThroughEnable	1	0x1	Writing 1 to this register enables pass-through mode. Writing 0 to this register disables pass-through mode thereby making the LBD compatible with PEC1.
0x10	PassThroughDotLength	16	0x0000	Number of dots for which pass-through mode will last. If the end of the line is reached first then passthrough will be disabled.
Work registers (need to be set up before processing a band)				
0x14	NextBandCurrReadAdr[21:5] (256-bit aligned DRAM address)	17	0x00000	Shadow register which is copied to *CurrReadAdr* when (*NextBandEnable* == 1 & *Go* == 0). *NextBandCurrReadAdr* is the address of the start of the next band of compressed bi-level data in DRAM.
0x18	NextBandLinesRemaining	15	0x0000	Shadow register which is copied to *LinesRemaining* when (*NextBandEnable* == 1 & *Go* == 0). *NextBandLinesRemaining* is the number of lines to be decoded in the next band of compressed bi-level data.

Table 107. LBD Configuration Registers

Address (LBD base+)	Register Name	#Bits	Value on Reset	Description
0x1C	NextBandPrevLineSource	1	0x0	Shadow register which is copied to *PrevLineSource* when *(NextBandEnable == 1 & Go == 0)*. 1 - use the previous line read from the SFU for decoding the first line at the start of the next band. 0 - ignore the previous line read from the SFU for decoding the first line at the start of the next band (an all 0's line is used instead).
0x20	NextBandEnable	1	0x0	If *(NextBandEnable == 1 & Go == 0)* then -*NextBandCurrReadAdr* is copied to *CurrReadAdr*, -*NextBandLinesRemaining* is copied to *LinesRemaining*, -*NextBandPrevLineSource* is copied to *PrevLineSource*, -*Go* is set, -*NextBandEnable* is cleared. To start LBD processing *NextBandEnable* should be set.
Work registers (read only for external access)				
0x24	CurrReadAdr[21:5] (256-bit aligned DRAM address)	17	-	The current 256-bit aligned read address within the compressed bi-level image (DRAM address). Read only register.
0x28	LinesRemaining	15	-	Count of number of lines remaining to be decoded. The band has finished when this number reaches 0. Read only register.
0x2C	PrevLineSource	1	-	1 - uses the previous line read from the SFU for decoding the first line at the start of the next band. 0 - ignores the previous line read from the SFU for decoding the first line at the start of the next band (an all 0's line is used instead). Read only register.
0x30	CurrWriteAdr	15	-	The current dot position for writing to the SFU. Read only register.
0x34	FirstLineOfBand	1	-	Indicates whether the current line is considered to be the first line of the band. Read only register.

24.3.3 Starting the LBD between bands

The LBD should be started *after* the SFU. The LBD is programed with a start address for the compressed bi-level data, a decode line length, the source of the previous line and a count of how many lines to decode. The LBD's *NextBandEnable* bit should then be set (this will set LBD *Go*). The LBD decodes a single band and then stops, clearing it's *Go* bit and issuing a pulse on *lbd_finishedband*. The LBD can then be restarted for the next band, while the HCU continues to process previously decoded bi-level data from the SFU.

There are 4 mechanisms for restarting the LBD between bands:

a. *lbd_finishedband* causes an interrupt to the CPU. The LBD will have stopped and cleared its *Go* bit. The CPU reprograms the LBD, typically the *NextBandCurrReadAdr, NextBandLines-Remaining* and *NextBandPrevLineSource* shadow registers, and sets *NextBandEnable* to restart the LBD.

b. The CPU programs the LBD's *NextBandCurrReadAdr, NextBandLinesRemaining,* and *Next-BandPrevLineSource* shadow registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the band the LBD clears *Go, NextBandEnable* is already set so the LBD restarts immediately.

c. The PCU is programmed so that *lbd_finishedband* triggers the PCU to execute commands from DRAM to reprogram the LBD's *NextBandCurrReadAdr, NextBandLinesRemaining,* and *Next-BandPrevLineSource* shadow registers and set *NextBandEnable* to restart the LBD. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

d. This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the LBD's *NextBandCurrReadAdr, NextBandLinesRemaining,* and *NextBandPrevLineSource* shadow registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the band the LBD clears *Go* and pulses *lbd_finishedband. NextBandEnable* is already set so the LBD restarts immediately. Simultaneously, *lbd_finishedband* triggers the PCU to fetch commands from DRAM. The LBD will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the LBD's shadow registers and sets *Next-BandEnable* for the next band.

24.3.4 Top-level Description

A block diagram of the LBD is shown in Figure 113.
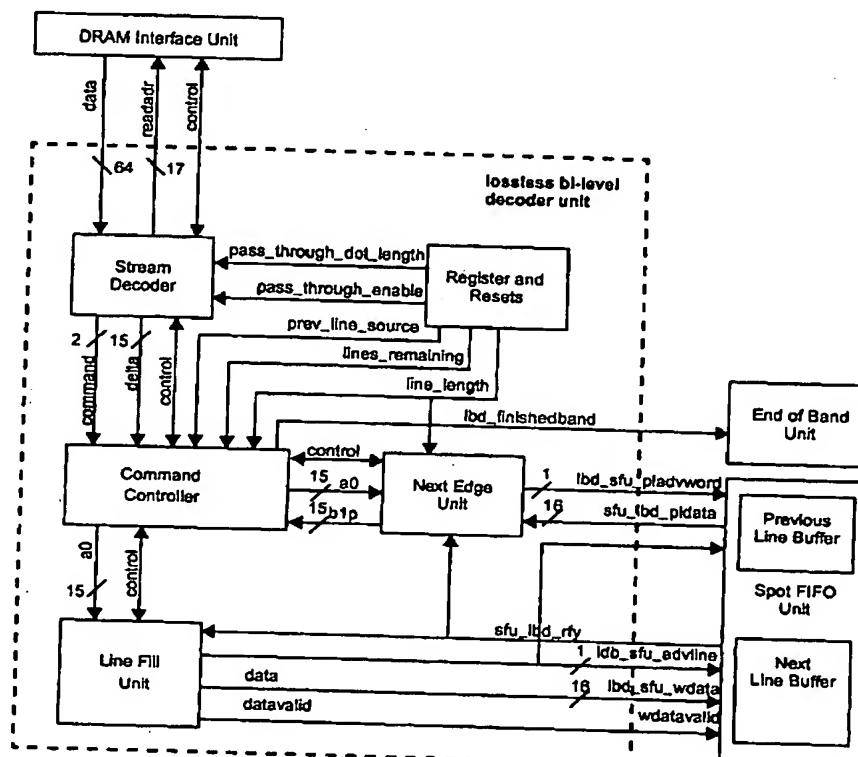
Figure 113. Block diagram of lossless bi-level decoder

The LBD contains the following sub-blocks:

Table 108. Functional sub-blocks in the LBD

name	description
Registers and Resets	PCU interface and configuration registers. Also generates the Go and the Reset signals for the rest of the LBD
Stream Decoder	Accesses the bi-level description from the DRAM through the DIU interface. It decodes the bit stream into a command with arguments, which it then passes to the command controller.
Command Controller	Interprets the command from the stream decoder and provide the line fill unit with a limit address and color to fill the SFU Next Line Buffer. It also provides the next edge unit starting address to look for the next edge.
Next Edge Unit	Scans through the Previous Line Buffer using its current address to find the next edge of a color provided by the command controller. The next edge unit outputs this as the next current address back to the command controller and sets a valid bit when this address is at the next edge.
Line Fill Unit	Fills the SFU Next Line Buffer with a color from its current address up to a limit address. The color and limit are provided by the command controller.

In the following description the LBD decodes data for its current decode line but writes this data into the SFU's *next* line buffer.

Naming of signals and logical blocks are taken from [18].

The LBD is able to stall mid-line should the SFU be unable to supply a previous line or receive a current line frame due to band processing latency.

All output control signals from the LBD must always be valid after reset. For example, if the LBD is not currently decoding, *lbd_sfu_advline* (to the SFU) and *lbd_finishedband* will always be 0.

24.3.5   Registers and Resets sub-block description

Since the CDU, LBD and TE all access the page band store, they share two registers that enable sequential memory accesses to the page band stores to be circular in nature. The CDU chapter lists these two registers. The register descriptions for the LBD are listed in Table 107.

During initialisation of the LBD, the *LineLength* and the *LinesRemaining* configuration values are written to the LBD. The '*Registers and Resets*' sub-block supplies these signals to the other sub-blocks in the LBD. In the case of *LinesRemaining*, this number is decremented for every line that is completed by the LBD.

If pass through is used during a band the *PassThroughEnable* register needs to be programmed and *PassThroughDotLength* programmed with the length of the compressed bits in pass through mode.

PrevLineSource is programmed during the initialisation of a band, if the previous line supplied for the first line is a valid previous line, a 1 is written to *PrevLineSource* so that the data is used. If a 0 is written the LBD ignores the previous line information supplied and acts as if it is receiving all zeros for the previous line regardless of what the out of the SFU is.

The '*Registers and Resets*' sub-block also generates the resets used by the rest of the LBD and the *Go* bit which tells the LBD that it can start requesting data from the DIU and commence decoding of the compressed data stream.

24.3.6   Stream Decoder Sub-block Description

The Stream Decoder reads the compressed bi-level image from the DRAM via the DIU (single accesses of 256-bits) into a double 256-bit FIFO. The barrel shift register uses the 64-bit word from the FIFO to fill up the empty space created by the barrel shift register as it is shifting it's contents. The bit stream is decoded into a command/arguments pair, which in turn is passed to the command controller.

A dataflow block diagram of the stream decoder is shown in Figure 114.
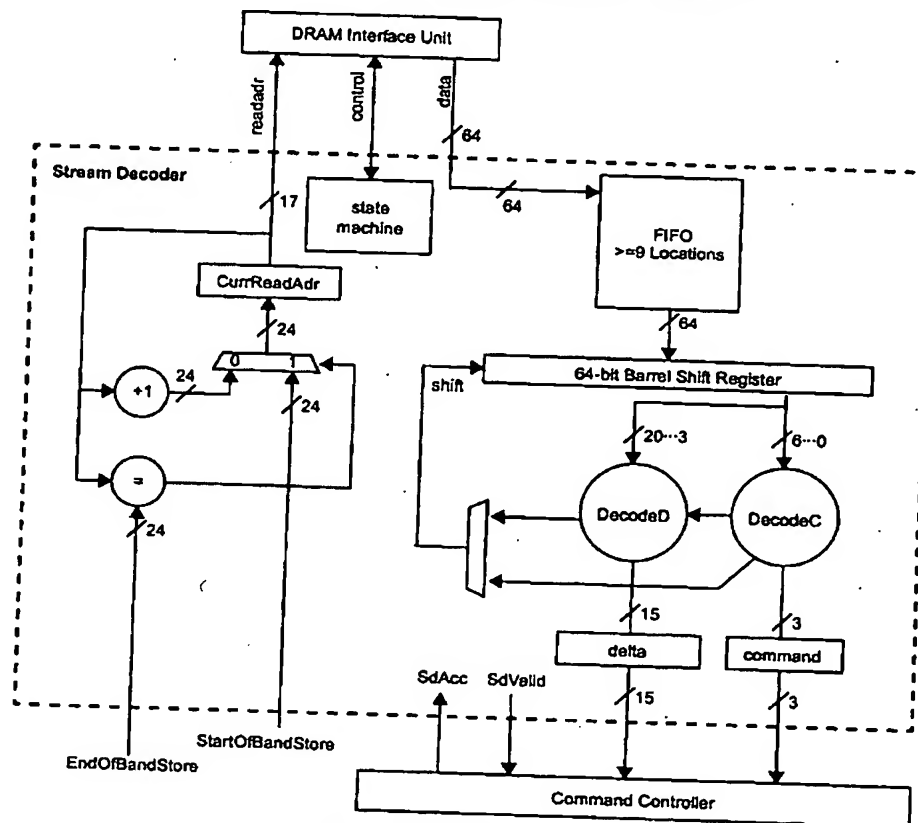
Figure 114. Stream decoder block diagram

24.3.6.1 DecodeC - Decode Command

The *DecodeC* logic encodes the command from bits 6..0 of the bit stream to output one of three commands: *SKIP, VERTICAL* and *RUNLENGTH*. It also provides an output to indicate how many bits were consumed, which feeds back to the barrel shift register.

There is a fourth command, *PASS_THROUGH*, which is not encoded in bits 6..0, instead it is inferred in a special runlength. If the stream decoder detects a short runlength value, i.e. a number less than 31, encoded as a medium runlength this tell the Stream Decoder that once the horizontal command containing this runlength is decoded completely the LBD enters *PASS_THROUGH* mode. Following the runlength there will be a number of bits that represent un-compressed data. The LBD will stay in *PASS_THROUGH* mode until all these bits have been decoded successfully, this will occur once a programmed number of bits is reached or the line ends, which ever comes first.

24.3.6.2 DecodeD - Decode Delta

The *DecodeD* logic decodes the run length from bits 20..3 of the bit stream. If *DecodeC* is decoding a vertical command, it will cause *DecodeD* to put constants of -3 through 3 on its output. The output *delta* is a 15 bit number, which is generally considered to be positive, but since it needs to only address to 13824 dots for an A4 page and 19488 dots for an A3 page (of 32,768), a 2's complement representation of -3,-2,-

1 will work correctly for the data pipeline that follows. This unit also outputs how many bits were consumed.

In the case of *PASS_THROUGH* mode, *DecodeD* parses the bits that represent the un-compressed data and this is used by the Line Fill Unit to construct the current line frame. *DecodeD* parses the bits at one bit per clock cycle and passes the bit in the less significant bit location of *delta* to the line fill unit.

DecodeD currently requires to know the color of the run length to decode it correctly as black and white runs are encoded differently. The stream decoder keeps track of the next color based on the current color and the current command.

24.3.6.3 State-machine

This state machine continuously fetches consecutive DRAM data whenever there is enough free space in the FIFO, thereby keeping the barrel shift register full so it can continually decode commands for the command controller. Note in Figure 114 that each read cycle *curr_read_addr* is compared to *end_of_band_store*. If the two are equal, *curr_read_addr* is loaded with *start_of_band_store* (circular memory addressing). Otherwise *curr_read_addr* is simply incremented. *start_of_band_store* and *end_of_band_store* need to be programed so that the distance between them is a multiple of the 256-bit DRAM word size.

When the state machine decodes a *SKIP* command, the state machine provides two *SKIP* instructions to the command controller.

The *RUNLENGTH* command has two different run lengths. The two run lengths are passed to the command controller as separate *RUNLENGTH* instructions. In the first instruction fetch, the first run length is passed, and the state machine selects the *DecodeD* shift value for the barrel shift. In the second instruction fetch from the command controller another *RUNLENGTH* instruction is generated and the respective shift value is decoded. This is achieved by forcing *DecodeC* to output a second *RUNLENGTH* instruction and the respective shift value is decoded.

For *PASS_THROUGH* mode, the *PASS_THROUGH* command is issued every time the command controller requests a new command. It does this until all the un-compressed bits have been processed.

24.3.7 Command Controller Sub-block Description

The Command Controller interprets the command from the Stream Decoder and provides the line fill unit with a limit address and color to fill the SFU Next Line Buffer. It provides the next edge unit with a starting address to look for the next edge and is responsible for detecting the end of line and generating the *eob_cc* signal that is passed to the line fill unit.

A dataflow block diagram of the command controller is shown in Figure 115. Note that data names such as *a0* and *b1p* are taken from [18], and they denote the reference or starting changing element on the coding line and the first changing element on the reference line to the right of *a0* and of the opposite color to *a0* respectively.
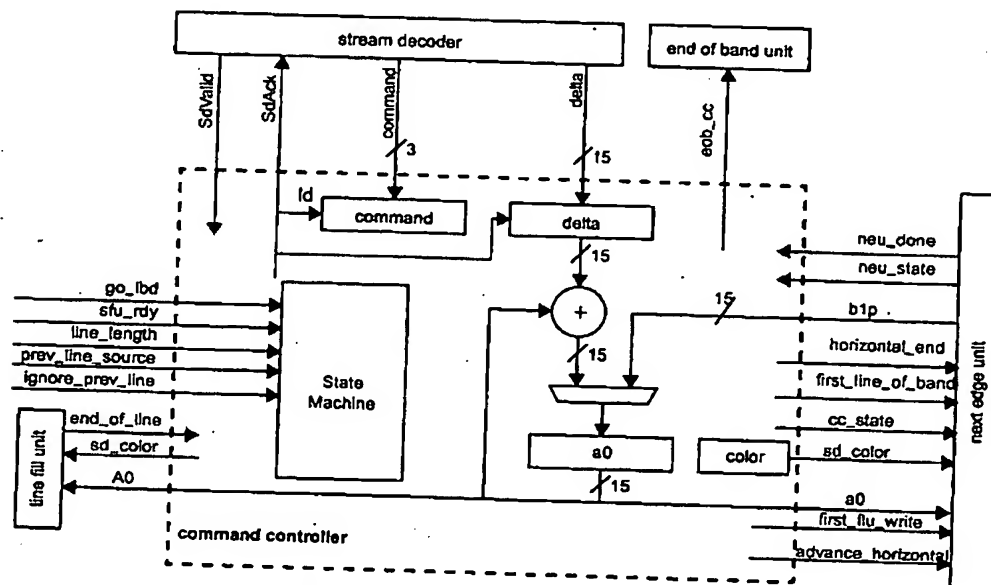
Figure 115. Command controller block diagram

24.3.7.1 State machine

The following is an explanation of all the states that the state machine utilizes.

i **START**

This is the state that the Command Controller enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state cannot be left until the reset has been removed, *Go* has been asserted and the *NEU* (Next Edge Unit), the *SD* (Stream Decoder) and the SFU are ready.

ii **AWAIT_BUFFER**

The *NEU* contains a buffer memory for the data it receives from the SFU. When the command controller enters this state the *NEU* detects this and starts buffering data, the command controller is able to leave this state when the state machine in the *NEU* has entered the *NEU_RUNNING* state. Once this occurs the command controller can proceed to the *PARSE* state.

iii **PAUSE_CC**

During the decode of a line it is possible for the FIFO in the stream decoder to get starved of data if the DRAM is not able to supply replacement data fast enough. Additionally the SFU can also stall mid-line due to band processing latency. If either of these cases occurs the LBD needs to pause until the stream decoder gets more of the compressed data stream from the DRAM or the SFU can receive or deliver new frames. All of the remaining states check if *sdvalid* goes to zero (this denotes a starving of the stream decoder) or if *sfu_lbd_rdy* goes to zero and that the LBD needs to pause. *PAUSE_CC* is the state that the command controller enters to achieve this and it does not leave this state until *sdvalid* and *sfu_lbd_rdy* are both asserted and the LBD can recommence decompressing.

iv **PARSE**

Once the command controller enters the PARSE state it uses the information that is supplied by the stream decoder. The first clock cycle of the state sees the *sdack* signal getting asserted informing the stream decoder that the current register information is being used so that it can fetch the next command.

When in this state the command controller can receive one of four valid commands:

a) Runlength or Horizontal

For this command the value given as delta is an integer that denotes the number of bits of the current color that must be added to the current line.

Should the current line position, $a0$, be added to the delta and the result be greater than the final position of the current frame being processed by the Line Fill Unit (only 16 bits at a time), it is necessary for the command controller to wait for the Line Fill Unit (LFU) to process up to that point. The command controller changes into the *WAIT_FOR_RUNLENGTH* state while this occurs.

When the current line position, $a0$, and the delta together equal or exceed the LINE_LENGTH, which is programmed during initialisation, then this denotes that it is the end of the current line. The command controller signals this to the rest of the LBD and then returns to the *START* state.

b) Vertical

When this command is received, it tells the command controller that, in the previous line, it needs to find a change from the current color to opposite of the current color, i.e. if the current color is white it looks from the current position in the previous line for the next time where there is a change in color from white to black. It is important to note that if a black to white change occurs first it is ignored.

Once this edge has been detected, the delta will denote which of the vertical commands to use, refer to Table 103. The delta will denote where the changing element in the current line is relative to the changing element on the previous line, for a Vertical(2) the new changing element position in the current line will correspond to the two bits extra from changing element position in the previous line.

Should the next edge not be detected in the current frame under review in the *NEU*, then the command controller enters the *WAIT_FOR_NE* state and waits there until the next edge is found.

c) Skip

A skip follow the same functionality as to *Vertical*(0) commands but the color in the current line is not changed as it is been filled out. The stream decoder supplies what looks like two separate skip commands that the command controller treats the same a two *Vertical*(0) commands and has been coded not to change the current color in this case.

d) Pass Through

When in pass through mode the stream decoder supplies one bit per clock cycle that is uses to construct the current frame. Once pass through mode is completed, which is controlled in the stream decoder, the LBD can recommence normal decompression again. The current color after pass through mode is the same color as the last bit in un-compressed data stream. Pass through mode does not need an extra state in the command controller as each pass through command received from the stream decoder can always be processed in one clock cycle.

v   *WAIT_FOR_RUNLENGTH*

As some *RUNLENGTH's* can carry over more than one 16-bit frame, this means that the Line Fill Unit needs longer than one clock cycle to write out all the bits represented by the *RUNLENGTH*. After the first clock cycle the command controller enters into the *WAIT_FOR_RUNLENGTH* state until all the RUN-LENGTH data has been consumed. Once finished and provided it is not the end of the line the command controller will return to the *PARSE* state.

vi   *WAIT_FOR_NE*

Similar to the *RUNLENGTH* commands the vertical commands can sometimes not find an edge in the current 16-bit frame. After the first clock cycle the command controller enters the *WAIT_FOR_NE* state and remains here until the edge is detected. Provided it is not the end of the line the command controller will return to the *PARSE* state.

vii FINISH_LINE

At the end of a line the command controller needs to hold its data for the SFU before going back to the START state. Command controller remains in the *FINISH_LINE* state for one clock cycle to achieve this.
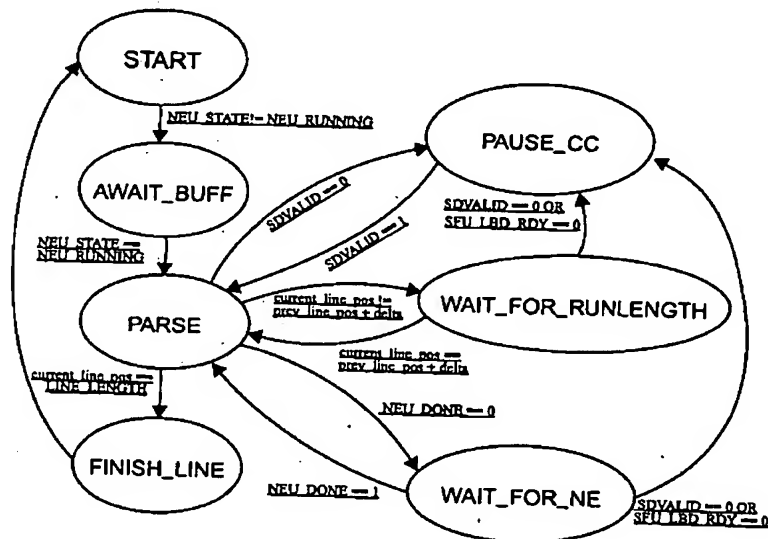

Figure 116. State diagram for the Command Controller (CC) state machine

24.3.8 Next Edge Unit Sub-block Description

The *Next Edge Unit* (*NEU*) is responsible for detecting color changes, or edges, in the previous line based on the current address and color supplied by the Command Controller. The *NEU* is the interface to the SFU and it buffers the previous line for detecting an edge. For an edge detect operation the Command Controller supplies the current address, this typically was the location of the last edge, but it could also be the end of a run length. With the current address a color is also supplied and using these two values the *NEU* will search the previous line for the next edge. If an edge is found the *NEU* returns this location to the Command Controller as the next address in the current line and it sets a valid bit to tell the Command Controller that the edge has been detected. The Line Fill Unit uses this result to construct the current line. The *NEU* operates on 16-bit words and it is possible that there is no edge in the current 16 bits in the *NEU*. In this case the NEU will request more words from the SFU and will keep searching for an edge. It will con-

tinue doing this until it finds an edge or reaches the end of the previous line, which is based on the *LINE_LENGTH*. A dataflow block diagram of the Next Edge unit is shown in Figure 117.
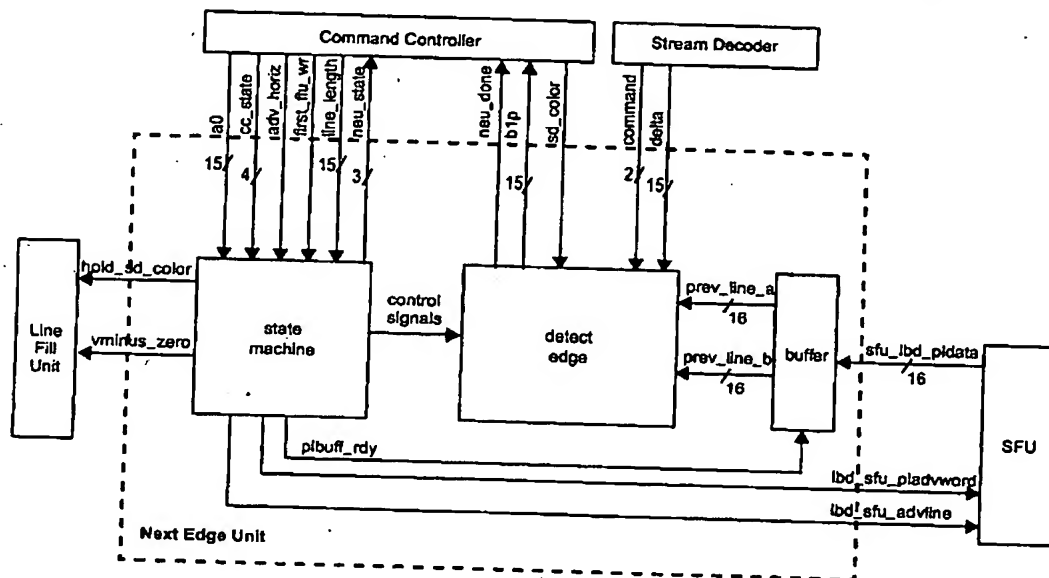
Figure 117. Next Edge Unit block diagram

24.3.8.1 NEU Buffer

The algorithm being employed for decompression is based on the whole previous line and is not delineated during the line. However the *Next Edge Unit, NEU*, can only receive 16 bits at a time from the SFU. This presents a problem for vertical commands if the edge occurs in the successive frame, but refers to a changing element in the current frame.

To accommodate this the *NEU* works on two frames at the same time, the current frame and the first 3 bits from the successive frame. This allows for the information that is needed from the previous line to construct the current frame of the current line.

In addition to this buffering there is also buffering right after the data is received from the SFU as the SFU output is not registered. The current implementation of the SFU takes two clock cycles from when a request for a current line is received until it is returned and registered. However when NEU requests a new

frame it needs it on the next clock cycle to maintain a decoded rate of 2 bits per clock cycle. A more detailed diagram of the buffer in the *NEU* is shown in Figure 118.
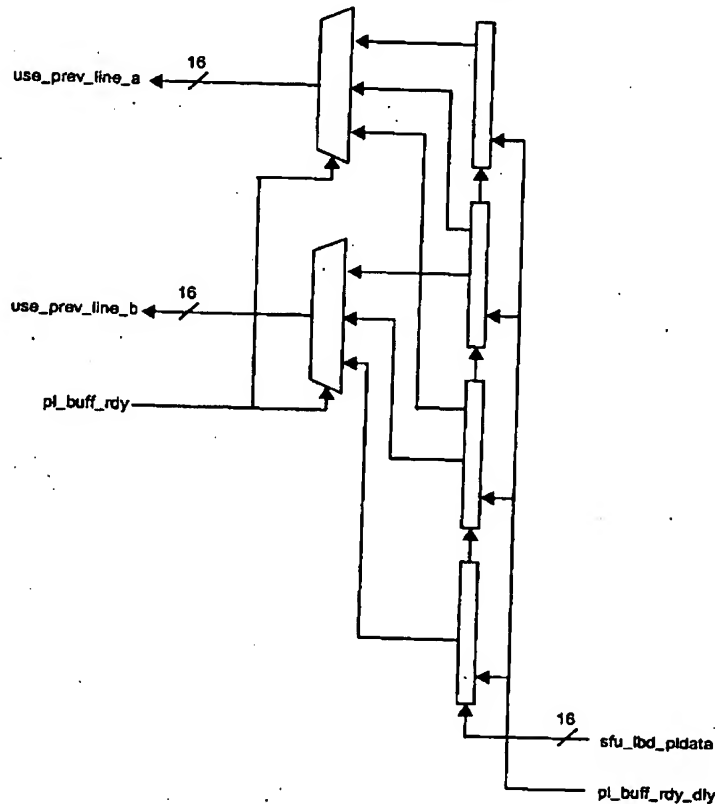
Figure 118. Next edge unit buffer diagram

The output of the buffer are two 16-bit vectors, *use_prev_line_a* and *use_prev_line_b*, that are used to detect an edge that is relevant to the current line being put together in the Line Fill Unit.

24.3.8.2 NEU Edge Detect

The *NEU* Edge Detect block takes the two 16 bit vectors supplied by the buffer and based on the current line position in the current line, *a0*, and the current color, *sd_color*, it will detect if there is an edge relevant

to the current frame. If the edge is found it supplies the current line position, *b1p*, to the command controller and the line fill unit. The configuration of the edge detect is shown in Figure 119.
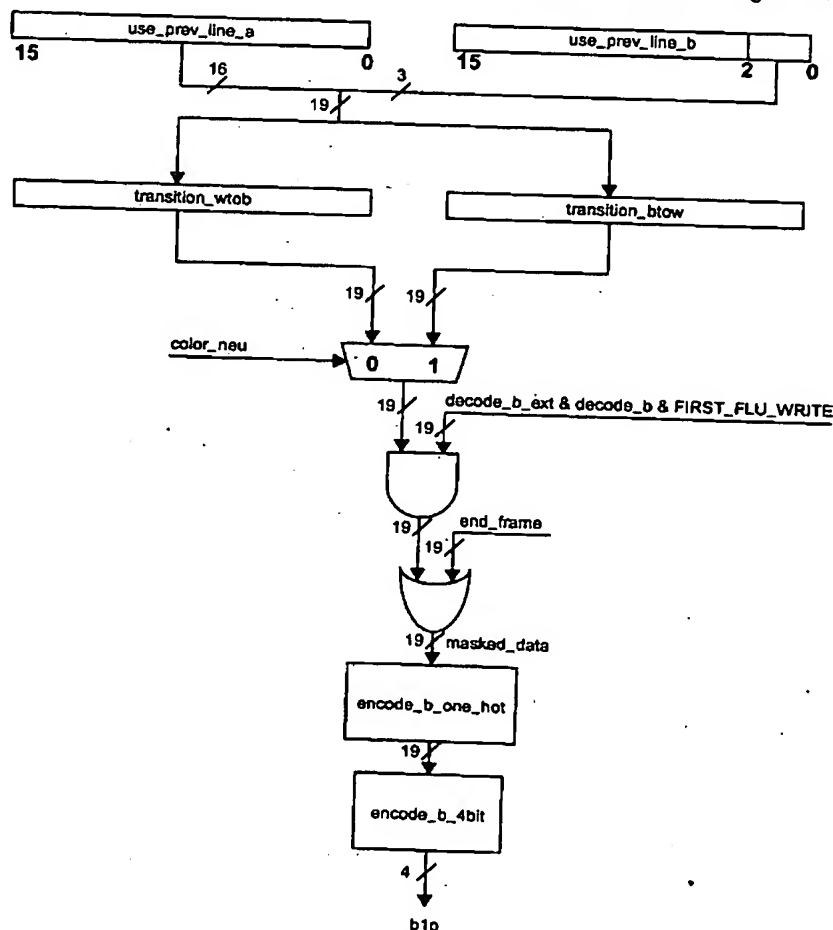
Figure 119. Next edge unit edge detect diagram

The two vectors from the buffer, *use_prev_line_a* and *use_prev_line_b*, pass into two sub-blocks, *transition_wtob* and *transition_btow*. *transition_wtob* detects if any white to black transitions occur in the 19 bit vector supplied and outputs a 19-bit vector displaying the transitions. *transition_wtob* is functionally the same as *transition_btow*, but it detects white to black transitions.

The two 19-bit vectors produced enter into a multiplexer and the output of the multiplexer is controlled by *color_neu*. *color_neu* is the current edge transition color that the edge detect is searching for.

The output of the multiplexer is masked against a 19-bit vector, the mask is comprised of three parts concatenated together: *decode_b_ext*, *decode_b* and *FIRST_FLU_WRITE*.

The output of *transition_wtob* (and it complement *transition_btow*) are all the transitions in the 16 bit word that is under review. The *decode_b* is a mask generated from *a0*. In bit-wise terms all the bits above and including *a0* are 1's and all bits below *a0* are 0's. When they are gated together it means that all the transitions below *a0* are ignored and the first transition after *a0* is picked out as the next edge.

The *decode_b* block decodes the 4 lsb of the current address (*a0*) into 16-bit mask bits that control which of the data bits are examined. Table 109 shows the truth table for this block.

Table 109. Decode_b truth table

input	output
0000	1111111111111111
0001	1111111111111110
0010	1111111111111100
0011	1111111111111000
0100	1111111111110000
0101	1111111111100000
0110	1111111111000000
0111	1111111110000000
1000	1111111100000000
1001	1111111000000000
1010	1111110000000000
1011	1111100000000000
1100	1111000000000000
1101	1110000000000000
1110	1100000000000000
1111	1000000000000000

For cases when there is a negative vertical command from the stream decoder it is possible that the edge is in the three lower significant bits of the next frame. The *decode_b_ext* block supplies the mask so that the necessary bits can be used by the *NEU* to detect an edge if present, Table 110 shows the truth table for this block.

Table 110. Decode_b_ext truth table

delta	output
Vertical(-3)	111
Vertical(-2)	111
Vertical(-1)	011
OTHERS	001

FIRST_FLU_WRITE is only used in the first frame of the current line. 2.2.5 a) in [18] refers to "Processing the first picture element", in which it states that "The first starting picture element, *a0*, on each coding line is imaginarily set at a position *just* before the first picture element, and is regarded as a white picture element". *transition_wtob* and *transition_btow* are set up produce this case for every single frame. However it is only used by the *NEU* if it is not masked out. This occurs when *FIRST_FLU_WRITE* is '1' which is only asserted at the beginning of a line.

2.2.5 b) in [18] covers the case of "Processing the last picture element", this case states that "The coding of the coding line continues until the position of the imaginary changing element situated after the last actual element is coded". This means that no matter what the current color is the *NEU* needs to always find an edge at the end of a line. This feature is used with negative vertical commands.

The vector, *end_frame*, is a "one-hot" vector that is asserted during the last frame. It asserts a bit in the end of line position, as determined by *LineLength*, and this simulates an edge in this location which is ORed with the transition's vector. The output of this, *masked_data*, is sent into the *encodeB_one_hot* block

24.3.8.3 Encode_b_one_hot

The *encode_b_one_hot* block is the first stage of a two stage process that encodes the data to determine the address of the 0 to 1 transition. Table 111 lists the truth table outlining the functionally required by this block.

Table 111. Encode_b_one_hot Truth Table

Input	Output
XXXXXXXXXXXXXXXXX1	00000000000000000001
XXXXXXXXXXXXXXXXX10	00000000000000000010
XXXXXXXXXXXXXXXX100	00000000000000000100
XXXXXXXXXXXXXXX1000	00000000000000001000
XXXXXXXXXXXXXX10000	00000000000000010000
XXXXXXXXXXXXX100000	00000000000000100000
XXXXXXXXXXXX1000000	00000000000001000000
XXXXXXXXXXX10000000	00000000000010000000
XXXXXXXXXX100000000	00000000000100000000
XXXXXXXXX1000000000	00000000001000000000
XXXXXXXX10000000000	00000000010000000000
XXXXXXX100000000000	00000000100000000000
XXXXXX1000000000000	00000001000000000000
XXXXX10000000000000	00000010000000000000
XXXX100000000000000	00001000000000000000
XXX1000000000000000	00010000000000000000
XX10000000000000000	00100000000000000000
X100000000000000000	01000000000000000000
1000000000000000000	10000000000000000000
0000000000000000000	00000000000000000000

The output of *encode_b_one_hot* is a "one-hot" vector that will denote where that edge transition is located. In cases of multiple edges, only the first one will be picked.

24.3.8.4 Encode_b_4bit

Encode_b_4bit is the second stage of the two stage process that encodes the data to determine the address of the 0 to 1 transition.

Encode_b_4bit receives the "one-hot" vector from *encode_b_one_hot* and determines the bit location that is asserted. If there is none present this means that there was no edge present in this frame. If there is a bit asserted the bit location in the vector is converted to a number, for example if bit 0 is asserted then the number is one, if bit one is asserted then the number is one, etc. The delta supplied to the *NEU* determines

what vertical command is being processed. The formula that is implemented to return blp to the command controller is:

$$\text{for } V(n)blp = x + n \text{ modulus16}$$

where x is the number that was extracted from the "one-hot" vector and n is the vertical command.

24.3.8.5 State machine
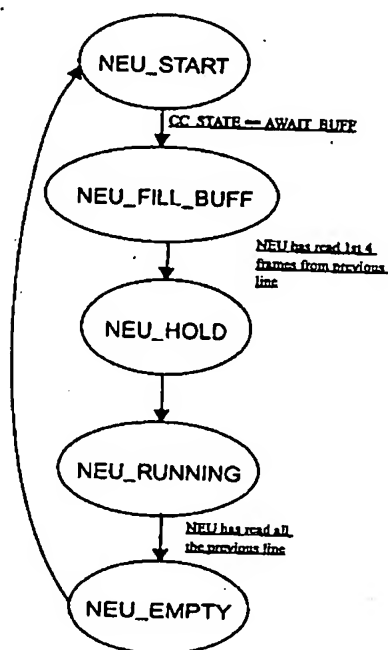
Figure 120. State diagram for the Next Edge Unit (NEU) state machine

The following is an explanation of all the states that the *NEU* state machine utilizes.

i   NEU_START

This is the state that NEU enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state can not left until the reset has been removed, *Go* has been asserted and it detects that the command controller has entered it's *AWAIT_BUFF* state. When this occurs the *NEU* enters the *NEU_FILL_BUFF* state.

ii   NEU_FILL_BUFF

Before any compressed data can be decoded the *NEU* needs to fill up its buffer with new data from the SFU. The rest of the LBD waits while the *NEU* retrieves the first four frames from the previous line. Once completed it enters the *NEU_HOLD* state.

iii   NEU_HOLD

The NEU waits in this state for one clock cycle while data requested from the SFU on the last access returns.

iv   NEU_RUNNING

NEU_RUNNING controls the requesting of data from the SFU for the remainder of the line by pulsing *lbd_sfu_pladvword* when the LBD needs a new frame from the SFU. When the *NEU* has received all the word it needs for the current line, as denoted by the *LineLength*, the NEU enters the *NEU_EMPTY* state.

v   NEU_EMPTY

NEU waits in this state while the rest of the LBD finishes outputting the completed line to the SFU. The NEU leaves this state when *Go* gets deasserted. This occurs when the *end_of_line* signal is detected from the LBD.

24.3.9 Line Fill Unit sub-block description

The Line Fill Unit, LFU, is responsible for filling the next line buffer in the SFU. The SFU receives the data in blocks of sixteen bits. The LFU uses the color and *a0* provided by the Command Controller and when it has put together a complete 16-bit frame, it is written out to the SFU. The LBD signals to the SFU that the data is valid by strobing the *lbd_sfu_wdatavalid* signal.

When the LFU is at the end of the line for the current line data it strobes *lbd_sfu_advline* to indicate to the SFU that the end of the line has occurred.

A dataflow block diagram of the line fill unit is shown in Figure 119.
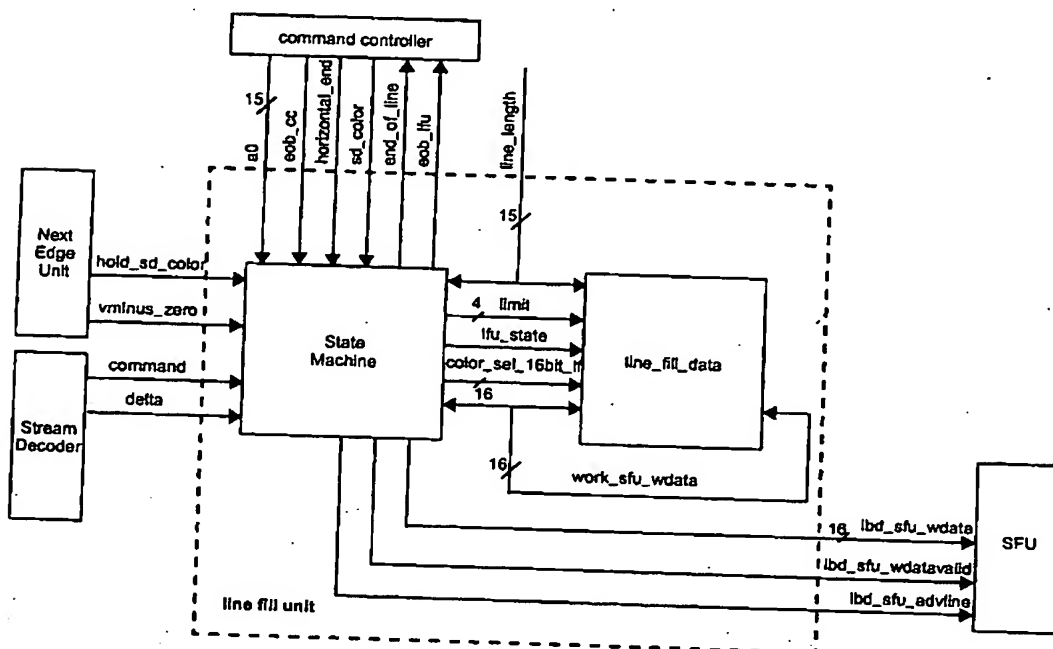
Figure 121. Line fill unit block diagram

The dataflow above has the following blocks:

24.3.9.1 State Machine

The following is an explanation of all the states that the LFU state machine utilizes.

i   *LFU_START*

This is the state that the LFU enters when a hard or soft reset occurs or when *Go* has been de-asserted. This state can not left until the reset has been removed, *Go* has been asserted and it detects that *a0* is no longer zero, this only occurs once the command controller start processing data from the *Next Edge Unit*, *NEU*.

ii   *LFU_NEW_REG*

LFU_NEW_REG is only entered at the beginning of a new frame. It can remain in this state on subsequent cycles if a whole frame is completed in one clock cycle. If the frame is completed the LFU will output the data to the SFU with the write enable signal. However if a frame is not completed in one clock cycle the state machine will change to the *LFU_COMPLETE_REG* state to complete the remainder of the frame. *LFU_NEW_REG* handles all the *lbd_sfu_wdata* writes and asserts *lbd_sfu_wdatavalid* as necessary.

iii LFU_COMPLETE_REG

LFU_COMPLETE_REG fills out all the remaining parts of the frame that were not completed in the first clock cycle. The command controller supplies the *a0* value and the color and the state machine uses these to derive the *limit* and *color_sel_16bit_lf* which the *line_fill_data* block needs to construct a frame. *Limit* is the four lower significant bits of *a0* and *color_sel_16bit_lf* is a 16-bit wide mask of *sd_color*. The state machine also maintains a check on the upper eleven bits of *a0*. If these increment from one clock cycle to the next that means that a frame is completed and the data can be written to the SFU. In the case of the *LineLength* being reached the Line Fill Unit fills out the remaining part of the frame with the color of the last bit in the line that was decoded.
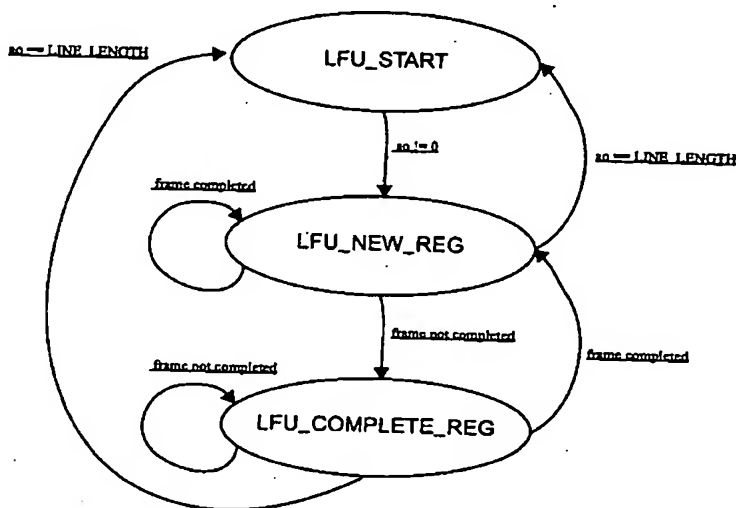
Figure 122. State diagram for the Line Fill Unit (LFU) state machine

24.3.9.2 line_fill_data

line_fill_data takes the *limit* value and the *color_sel_16bit_lf* values and constructs the current frame that the command controller and the next edge unit are decoding. The following pseudo code illustrate the logic followed by the line_fill_data. *work_sfu_wdata* is exported by the LBD to the SFU as *lbd_sfu_wdata*.

```
if (lfu_state == LFU_START) OR (lfu_state == LFU_NEW_REG) then
    work_sfu_wdata = color_sel_16bit_lf
else
    work_sfu_wdata[(15 - limit) downto limit] =
            color_sel_16bit_lf[(15 - limit) downto limit]
```

25 Spot FIFO Unit (SFU)

25.1 OVERVIEW

The Spot FIFO Unit (SFU) provides the means by which data is transferred between the LBD and the HCU. By abstracting the buffering mechanism and controls from both units, the interface is clean between the data user and the data generator. The amount of buffering can also be increased or decreased without affecting either the LBD or HCU. Scaling of data is performed in the horizontal and vertical directions by the SFU so that the output to the HCU matches the printer resolution. Non-integer scaling is supported in both the horizontal and vertical directions. Typically, the scale factor will be the same in both directions but may be programmed to be different.

25.2 MAIN FEATURES OF THE SFU

The SFU replaces the Spot Line Buffer Interface (SLBI) in PEC1. The spot line store is now located in DRAM.

The SFU outputs the previous line to the LBD, stores the next line produced by the LBD and outputs the HCU read line. Each interface to DRAM is via a feeder FIFO. The LBD interfaces to the SFU with a data width of 16 bits. The SFU interfaces to the HCU with a data width of 1 bit.

Since the DRAM word width is 256-bits but the LBD line length is a multiple of 16 bits, a capability to flush the last multiples of 16-bits at the end of a line into a 256-bit DRAM word size is required. Therefore, SFU reads of DRAM words at the end of a line, which do not fill the DRAM word, will already be padded.

A signal *sfu_ldb_rdy* to the LBD indicates that the SFU is available for writing and reading. For the first LBD line after SFU *Go* has been asserted, previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD (zero data is supplied instead), and *sfu_ldb_rdy* to the LBD indicates that the SFU is available for writing. *lbd_sfu_advline* tells the SFU to advance to the next line. *lbd_sfu_pladvword* tells the SFU to supply the next 16-bits of previous line data. Until the number of *lbd_sfu_pladvword* strobes received is equivalent to the LBD line length, *sfu_ldb_rdy* indicates that the SFU is available for both reading and writing. Thereafter it indicates the SFU is available for writing. The LBD should not generate *lbd_sfu_pladvword* or *lbd_sfu_advline* strobes until *sfu_ldb_rdy* is asserted.

A signal *sfu_hcu_avail* indicates that the SFU has data to supply to the HCU. Another signal *hcu_sfu_advdot*, from the HCU, tells the SFU to supply the next dot. The HCU should not generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

X and Y non-integer scaling of the bi-level dot data is performed in the SFU.

At 1600 dpi the SFU requires 1 dot per cycle for all DRAM channels, 3 dots per cycle in total (read + read + write). Therefore the SFU requires two 256 bit read DRAM access per 256 cycles, 1 write access every 256 cycles. A single DIU read interface will be shared for reading the current and previous lines from DRAM.

25.3   BI-LEVEL DRAM MEMORY BUFFER BETWEEN LBD, SFU AND HCU
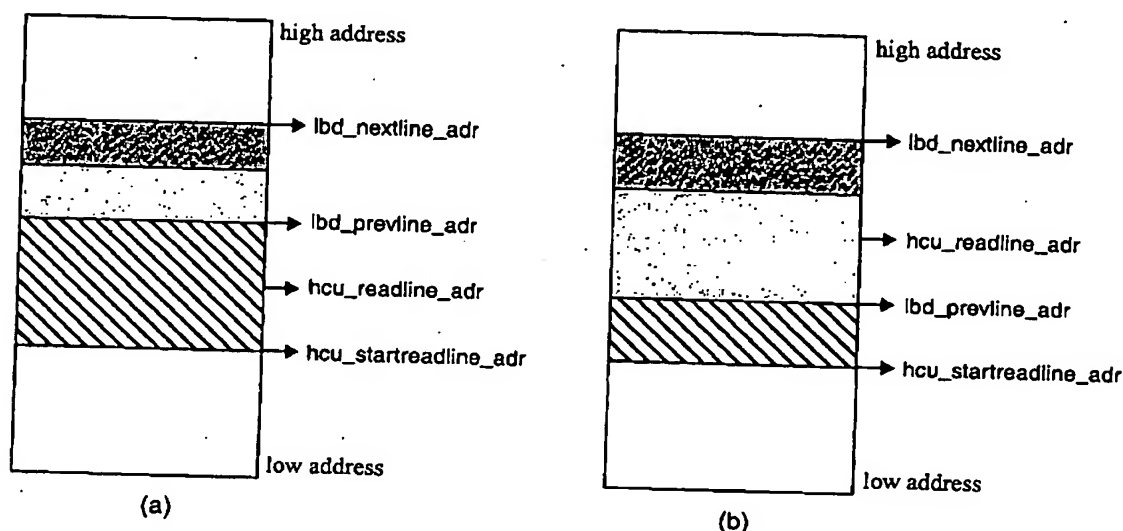
Key: ☐ Free buffer space

▨ Filled buffer space accessed by LBD Interface FIFOs

◩ Filled Buffer space read by HCU Read Line FIFO

☐ Filled Buffer space read by both HCU Read Line FIFO and LBD Interface FIFOs

Figure 123. Bi-level DRAM buffer

Figure shows a bi-level buffer store in DRAM. Figure (a) shows the LBD previous line address reading after the HCU read line address in DRAM. Figure (b) shows the LBD previous line address reading before the HCU read line address in DRAM.

Although the LBD and HCU read and write complete lines of data, the bi-level DRAM buffer is not line based. The buffering between the LBD, SFU and HCU is a FIFO of programmable size. The only line based concept is that the line the HCU is currently reading cannot be over-written because it may need to be re-read for scaling purposes.

The SFU interfaces to DRAM via three FIFOs:

  a. The *HCUReadLineFIFO* which supplies dot data to the HCU.

  b. The *LBDNextLineFIFO* which writes decompressed bi-level data from the LBD.

  c. The *LBDPrevLineFIFO* which reads previous decompressed bi-level data for the LBD.

There are four address pointers used to manage the bi-level DRAM buffer:

  a. $hcu\_readline\_adr[21:5]$ is the read address in DRAM for the *HCUReadLineFIFO*.

  b. $hcu\_startreadline\_adr[21:5]$ is the start address in DRAM for the current line being read by the *HCUReadLineFIFO*.

c. *lbd_nextline_adr[21:5]* is the write address in DRAM for the *LBDNextLineFIFO*.

d. *lbd_prevline_adr[21:5]* is the read address in DRAM for the *LBDPrevLineFIFO*.

The address pointers must obey certain rules which indicate whether they are valid:

a. *hcu_readline_adr[21:5]* is only valid if it is reading earlier in the line than *lbd_nextline_adr[21:5]* is writing i.e. *hlf_adrvalid = hcu_readline_adr[21:5] != lbd_nextline_adr[21:5]*.

b. The SFU cannot overwrite the current line that the HCU is reading from i.e. *hlf_startadrvalid = lbd_nextline_adr[21:5] != hcu_startreadline_adr[21:5]*.

c. The *LBDNextLineFIFO* must be writing earlier in the line than *LBDPrevLineFIFO* is reading and must not overwrite the current line that the HCU is reading from i.e. *nlf_adrvalid = lbd_nextline_adr[21:5] != lbd_prevline_adr[21:5] AND hcu_startreadline_valid*.

d. The *LBDPrevLineFIFO* can read right up to the address that *LBDNextLineFIFO* is writing i.e. *plf_adrvalid = lbd_prevline_adr[21:5] != lbd_nextline_adr[21:5]*.

e. At startup i.e. when *sfu_go* is asserted, the pointers are reset to *start_sfu_adr[21:5]*. The first LBD *NextLineFIFO* data is allowed to be written to *lbd_nextline_adr[21:5]* even though *nlf_adrvalid* is initially invalid.

f. The address pointers can wrap around the SFU bi-level store area in DRAM.

As a guideline, the typical FIFO size should be a minimum of 2 lines stored in DRAM, nominally 3 lines, up to a programmable number of lines. A larger buffer allows lines to be decompressed in advance. This can be useful for absorbing local complexities in compressed bi-level images.

25.4 DRAM ACCESS REQUIREMENTS

The SFU has 1 read interface to the DIU and 1 write interface. The read interface is shared between the previous and current line read FIFOs.

The spot line store requires 5.1 Kbytes of DRAM to store 3 A4 lines. The SFU will read and write the spot line store in single 256-bit DRAM accesses. The SFU will need 256-bit double buffers for each of its previous, current and next line interfaces.

The SFU's DIU bandwidth requirements are summarized in Table 112.

Table 112. DRAM bandwidth requirements

Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth required to be supported by DIU (bits/cycle)	Average Bandwidth (bits/cycle)
Read	128[1]	2	2
Write	256[2]	1	1

1: Two separate reads of 1 bit/cycle.

2: Write at 1 bit/cycle.

25.5 SCALING

Scaling of bi-level data is performed in both the horizontal and vertical directions by the SFU so that the output to the HCU matches the printer resolution. The SFU supports non-integer scaling with the scale factor represented by a numerator and a denominator. Only scaling up of the bi-level data is allowed, i.e. the numerator should be greater than or equal to the denominator. Scaling is implemented using a counter

as described in the pseudocode below. An advance pulse is generated to move to the next dot (x-scaling) or line (y-scaling).

```
if (count + denominator >= numerator) then
    count = (count + denominator) - numerator
    advance = 1
else
    count = count + denominator
    advance = 0
```

X scaling controls whether the SFU supplies the next dot or a copy of the current dot when the HCU asserts *hcu_sfu_advdot*. The SFU counts the number of *hcu_sfu_advdot* signals from the HCU. When the SFU has supplied an entire HCU line of data, the SFU will either re-read the current line from DRAM or advance to the next line of HCU read data depending on the programmed Y scale factor.

An example of scaling for *numerator* = 7 and *denominator* = 3 is given in Table 113. The signal *advance* if asserted causes the next input dot to be output on the next cycle, otherwise the same input dot is output

Table 113. Non-Integer scaling example for scaleNum = 7, scaleDenom = 3

count	advance	dot
0	0	1
3	0	1
6	1	1
2	0	2
5	1	2
1	0	3
4	1	3
0	0	4
3	0	4
6	1	4
2	0	5

25.6 LEAD-IN AND LEAD-OUT CLIPPING

To account for the case where there may be two SoPEC devices, each generating its own portion of a dot-line, the first dot in a line may not be replicated the total scale-factor number of times by an individual SoPEC. The dot will ultimately be scaled-up correctly with both devices doing part of the scaling, one on its lead-out and the other on its lead in. Scaled up dots on the lead-out, i.e. which go beyond the HCU line-length, will be ignored. Scaling on the lead-in, i.e. of the first valid dot in the line, is controlled by setting the *XstartCount* register.

At the start of each line *count* in the pseudo-code above is set to *XstartCount*. If there is no lead-in, *XstartCount* is set to 0 i.e. the first value of *count* in Table 113. If there is lead-in then *XstartCount* needs to be set to the appropriate value of *count* in the sequence above.

25.7 INTERFACES BETWEEN LDB, SFU AND HCU
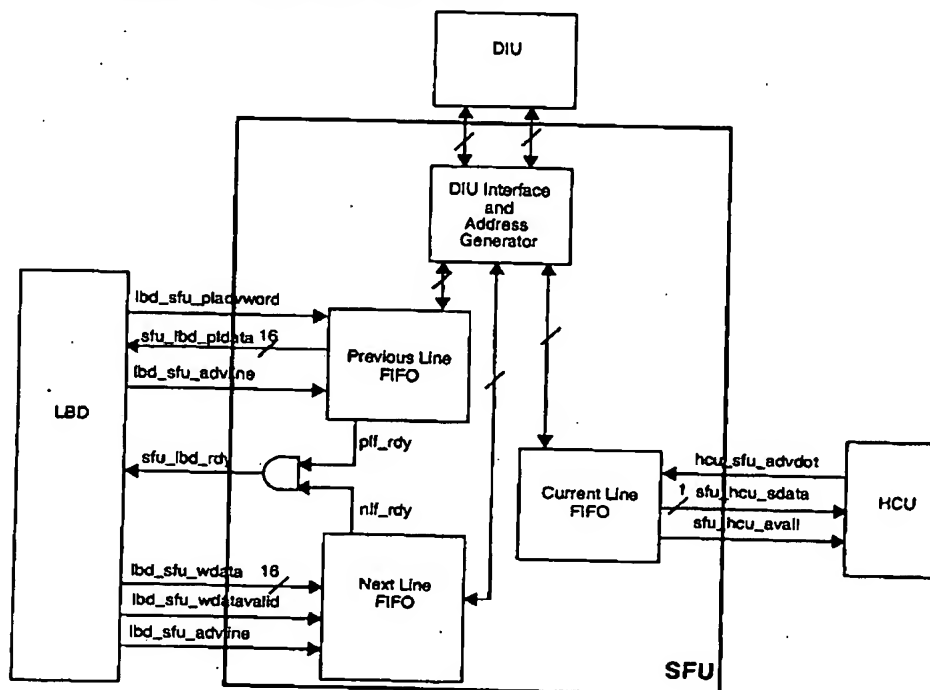
Figure 124. Interfaces between LBD/SFU/HCU

25.7.1 LDB-SFU Interfaces

The LBD has two interfaces to the SFU. The LBD writes the next line to the SFU and reads the previous line from the SFU.

25.7.1.1 *LBDNextLineFIFO Interface*

The *LBDNextLineFIFO* interface from the LBD to the SFU comprises the following signals:
- *lbd_sfu_wdata*, 16-bit write data.
- *lbd_sfu_wdatavalid*, write data valid.
- *lbd_sfu_advline*, signal indicating LDB has advanced to the next line.

The LBD should not write to the SFU until *sfu_lbd_rdy* is true. The LBD can therefore stall waiting for the *sfu_lbd_rdy* signal.

25.7.1.2 *LBDPrevLineFIFO Interface*

The *LBDPrevLineFIFO* interface from the SFU to the LBD comprises the following signals:
- *sfu_lbd_pldata*, 16-bit data.

The previous line read buffer interface from the LBD to the SDU comprises the following signals:
- *lbd_sfu_pladvword*, signal indicating to the SFU to supply the next 16-bit word.
- *lbd_sfu_advline*, signal indicating LDB has advanced to the next line.

Previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD (zero data is supplied instead). The LBD should not assert *lbd_sfu_pladvword* unless *sfu_lbd_rdy* is asserted.

25.7.1.3 Common Control Signals

sfu_ldb_rdy indicates to the LBD that the SFU is available for writing. After the first *lbd_sfu_advline* and before the number of *lbd_sfu_pladvword* strobes received is equivalent to the LBD line length, *sfu_ldb_rdy* indicates that the SFU is available for both reading and writing. Thereafter it indicates the SFU is available for writing.

The LBD should not generate *lbd_sfu_pladvword* or *lbd_sfu_advline* strobes until *sfu_ldb_rdy* is asserted.

25.7.2 SFU-HCU Current Line FIFO Interface

The interface from the SFU to the HCU comprises the following signals:
- *sfu_hcu_sdata*, 1-bit data.
- *sfu_hcu_avail*, data valid signal indicating that there is data available in the SFU *HCUReadLineFIFO*.

The interface from HCU to SFU comprises the following signals:
- *hcu_sfu_advdot*, indicating to the SFU to supply the next dot.

The HCU should not generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

25.8   IMPLEMENTATION

25.8.1   Definitions of IO

Table 114. SFU Port List

Port Name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
DIU Read Interface signals			
sfu_diu_rreq	1	Out	SFU requests DRAM read. A read request must be accompanied by a valid read address.
sfu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_sfu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *sfu_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to SoPEC Units. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits are bits 255:192 of 256 bit word.
diu_sfu_rvalid	1	In	Signal from DIU telling SoPEC Unit that valid read data is on the *diu_data* bus.
DIU Write Interface signals			
sfu_diu_wreq	1	Out	SFU requests DRAM write. A write request must be accompanied by a valid write address together with valid write data and a write valid.
sfu_diu_wadr[21:5]	17	Out	Write address to DIU 17 bits wide (256-bit aligned word).
diu_sfu_wack	1	In	Acknowledge from DIU that write request has been accepted and new write address can be placed on *sfu_diu_wadr*.
sfu_diu_data[63:0]	64	Out	Data from SFU to DIU. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits are bits 255:192 of 256 bit word.
sfu_diu_wvalid	1	Out	Signal from PEP Unit indicating that data on *sfu_diu_data* is valid.
PCU Interface data and control signals			
pcu_addr[5:2]	4	In	PCU address bus. Only 4 bits are required to decode the address space for this block
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU
sfu_pcu_datain[31:0]	32	Out	Read data bus from the SFU to the PCU
pcu_rwn	1	In	Common read/not-write signal from the PCU
pcu_sfu_sel	1	In	Block select from the PCU. When *pcu_sfu_sel* is high both *pcu_addr* and *pcu_dataout* are valid

Table 114. SFU Port List

Port Name	Pins	I/O	Description
sfu_pcu_rdy	1	Out	Ready signal to the PCU. When *sfu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *sfu_pcu_datain* is valid.
LBD Interface Data and Control Signals			
sfu_lbd_rdy	1	Out	Signal indication that SFU has previous line data available and is ready to be written to.
lbd_sfu_advline	1	In	Line advance signal for both next and previous lines.
lbd_sfu_pladvword	1	In	Advance word signal for previous line buffer.
sfu_ldb_pldata[15:0]	16	Out	Data from the previous line buffer.
lbd_sfu_wdata[15:0]	16	In	Write data for next line buffer.
lbd_sfu_wdatavalid	1	In	Write data valid signal for next line buffer data.
HCU Interface Data and Control Signals			
hcu_sfu_advdot	1	In	Signal indicating to the SFU that the HCU is ready to accept the next dot of data from SFU.
sfu_hcu_sdata	1	Out	Bi-level dot data.
sfu_hcu_avail	1	Out	Signal indicating valid bi-level dot data on *sfu_hcu_sdata*.

25.8.2 Configuration Registers

Table 115. SFU Configuration Registers

Address (SFU base)	register name	bits	value on reset	description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the SFU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0x0	Writing 1 to this register starts the SFU. Writing 0 to this register halts the SFU. When Go is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When Go is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The SFU must be started before the LBD is started. This register can be read to determine if the SFU is running (1 - running, 0 - stopped).
Setup registers (constant for during processing the page)				
0x08	HCUNumDots	16	0x0000	Width of HCU line (in dots).
0x0C	HCUDRAMWords	8	0x00	Number of 256-bit DRAM words in a HCU line.
0x10	LBDNumWords	12	0x000	Number of 16-bit words in an LBD line. (LBD line length must be a multiple of 16 bits).
0x14	StartSfuAdr[21:5] (256-bit aligned DRAM address)	17	0x00000	First SFU location in memory.
0x18	EndSfuAdr[21:5] (256-bit aligned DRAM address)	17	0x00000	Last SFU location in memory.
0x1C	XstartCount	8	0x00	Value to be loaded at the start of every line into the counter used for scaling in the X direction. Used to control the scaling of the first dot in a line. This value will typically equal zero, except in the case where a number of dots are clipped on the lead in to a line.
0x20	XscaleNum	8	0x01	Numerator of spot data scale factor in X direction.
0x24	XscaleDenom	8	0x01	Denominator of spot data scale factor in X direction.
0x28	YscaleNum	8	0x01	Numerator of spot data scale factor in Y direction.
0x2C	YscaleDenom	8	0x01	Denominator of spot data scale factor in Y direction.
Work registers (PCU has read-only access)				

Table 115. SFU Configuration Registers

Address (SFUbase +)	register name	#bits	value on reset	description
0x30	HCUReadLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Current address pointer in DRAM to HCU read data. Read only register.
0x34	HCUStartReadLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Start address in DRAM of line being read by HCU buffer in DRAM. Read only register.
0x38	LBDNextLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Current address pointer in DRAM to LBD write data. Read only register
0x3C	LBDPrevLineAdr[21:5] (256-bit aligned DRAM address)	17	-	Current address pointer in DRAM to LBD read data. Read only register

25.8.3   SFU sub-block partition
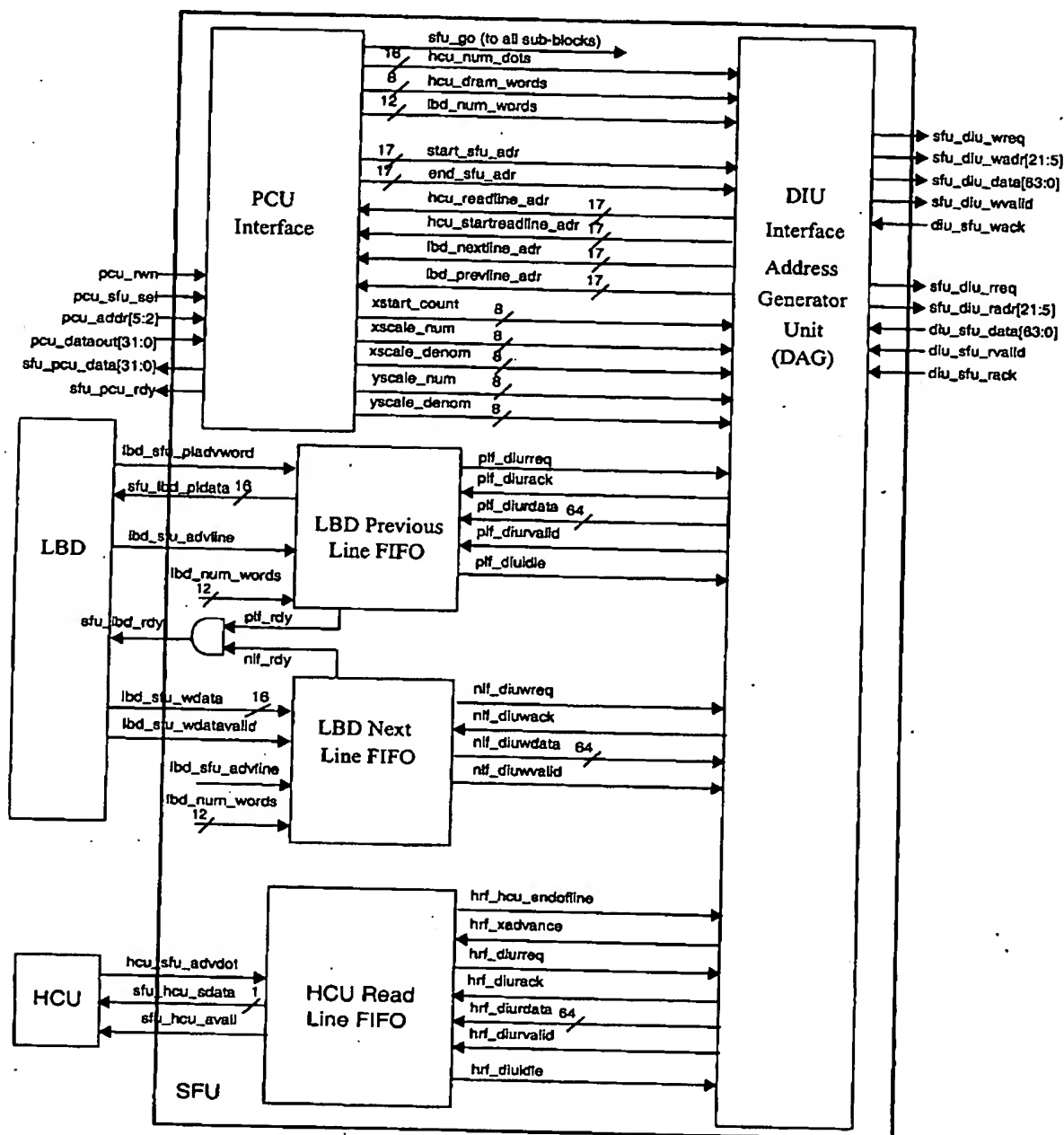
Figure 125. SFU Sub-Block Partition

The SFU contains a number of sub-blocks:

name	description
PCU Interface	PCU interface, configuration and status registers. Also generates the *Go* and the *Reset* signals for the rest of the SFU
LBD Previous Line FIFO	Contains FIFO which is read by the LBD previous line interface.
LBD Next Line FIFO	Contains FIFO which is written by the LBD next line interface.
HCU Read Line FIFO	Contains FIFO which is read by the HCU interface.
DIU Interface and Address Generator	Contains DIU read interface and DIU write interface. Manages the address pointers for the bi-level DRAM buffer. Contains X and Y scaling logic.

The various FIFO sub-blocks have no knowledge of where in DRAM their read or write data is stored. In this sense the FIFO sub-blocks are completely de-coupled from the bi-level DRAM buffer. All DRAM address management is centralised in the DIU Interface and Address Generation sub-block. DRAM access is pre-emptive i.e. after a FIFO unit has made an access then as soon as the FIFO has space to read or data to write a DIU access will be requested immediately. This ensures there are no unnecessary stalls introduced e.g. at the end of an LBD or HCU line.

There now follows a description of the SFU sub-blocks.

25.8.4   PCU Interface Sub-block

The PCU interface sub-block provides for the CPU to access SFU specific registers by reading or writing to the SFU address space.

25.8.5 LBDPrevLineFIFO sub-block

Table 116: LBDPrevLineFIFO Additional IO Definitions

Port Name	Pins	I/O	Description
Internal Output			
plf_rdy	1	Out	Signal indicating *LBDPrevLineFIFO* is ready to be read from. Until the first *lbd_sfu_advline* for a band has been received and after the number of *lbd_sfu_pladvword* strobes received for a line is equal to *LBDNumWords*, plf_rdy is always asserted. During the second and subsequent lines *plf_rdy* is deasserted whenever the *LBDPrevLineFIFO* is empty.
DIU and Address Generation sub-block Signals			
plf_diurreq	1	Out	Signal indicating the *LBDPrevLineFIFO* has 256-bits of data free.
plf_diurack	1	In	Acknowledge that read request has been accepted and *plf_diurreq* should be de-asserted.
plf_diurdata	1	In	Data from the DIU to *LBDPrevLineFIFO*. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits is are 255:192 of 256 bit word.
plf_diurvalid	1	In	Signal indicating data on *plf_diurdata* is valid.
plf_diuidle	1	Out	Signal indicating DIU state-machine is in the IDLE state.

25.8.5.1 General Description

The *LBDPrevLineFIFO* sub-block comprises a double 256-bit buffer between the LBD and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the DIU Interface and Address Generator sub-block and read by the LBD.
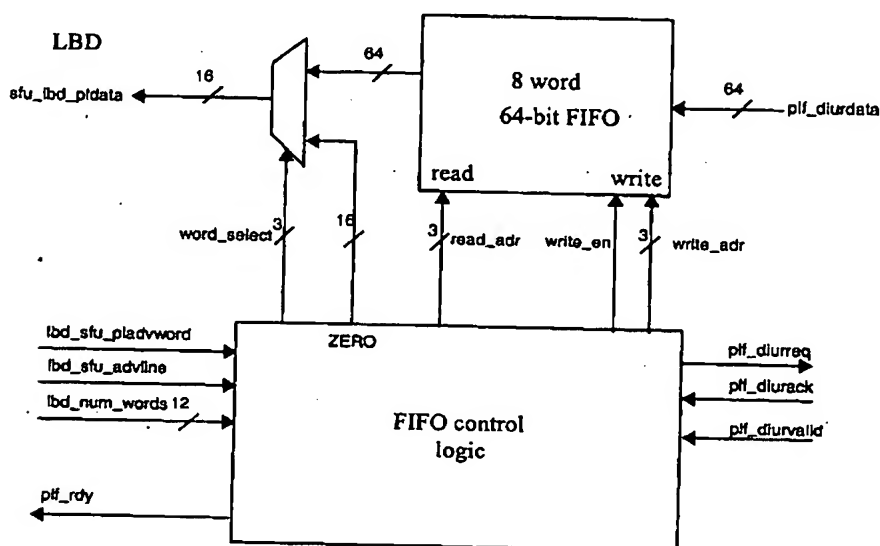
Figure 126. LBDPrevLineFifo Sub-block

Whenever 4 locations in the FIFO are free the FIFO will request 256-bits of data from the DIU Interface and Address Generation sub-block by asserting *plf_diurreq*. A signal *plf_diurack* indicates that the request has been accepted and *plf_diurreq* should be de-asserted.

The data is written to the FIFO as 64-bits on *plf_diurdata[63:0]* over 4 clock cycles. The signal *plf_diurvalid* indicates that the data returned on *plf_diurdata[63:0]* is valid. *plf_diurvalid* is used to generate the FIFO write enable, *write_en*, and to increment the FIFO write address, *write_adr[2:0]*. If the *LBD-PrevLineFIFO* still has 256-bits free then *plf_diurreq* should be asserted again.

The DIU Interface and Address Generation sub-block handles all address pointer management and DIU interfacing and decides whether to acknowledge a request for data from the FIFO.
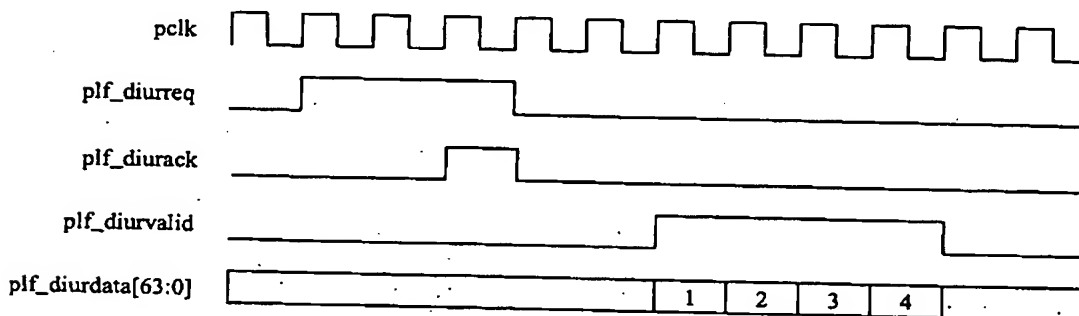
Figure 127. Timing of signals on the LBDPrevLineFIFO interface to DIU and Address Generator

The state diagram of the *LBDPrevLineFIFO* DIU Interface is shown in Figure 128. If *sfu_go* is deasserted then the state-machine returns to its *idle* state.
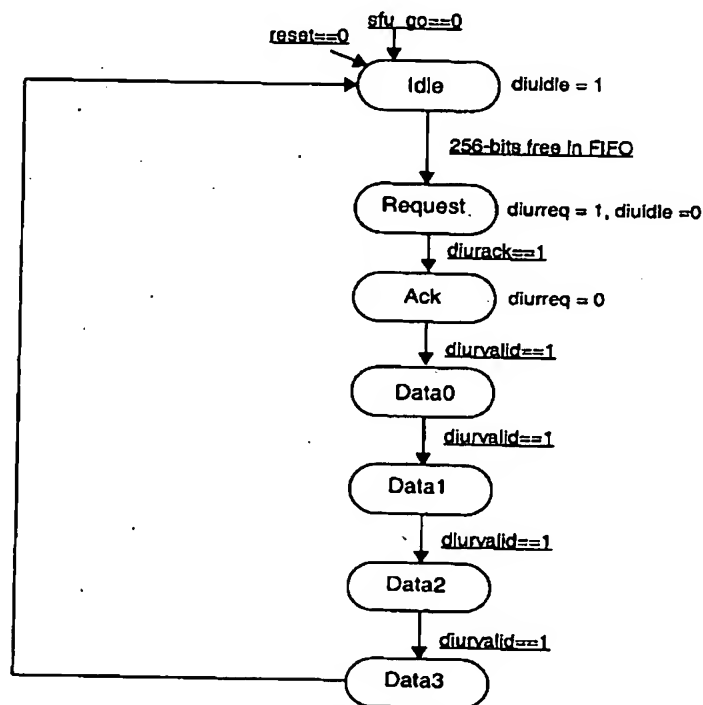
Figure 128. Timing of signals on LBDPrevLineFIFO interface to DIU and Address Generator

The LBD reads 16-bit wide data from the *LBDPrevLineFIFO* on *sfu_lbd_pldata[15:0]*. *lbd_sfu_pladvword* from the LBD tells the *LBDPrevLineFIFO* to supply the next 16-bit word. The FIFO control logic generates a signal *word_select* which selects the next 16-bits of the 64-bit FIFO word to output on *sfu_lbd_pldata[15:0]*. When the entire current 64-bit FIFO word has been read by the LBD *lbd_sfu_pladvword* will cause the next word to be popped from the FIFO.

Previous line data is not supplied until after the first *lbd_sfu_advline* strobe from the LBD after *sfu_go* is asserted (zero data is supplied instead). Until the first *lbd_sfu_advline* strobe after *sfu_go* *lbd_sfu_pladvword* strobes are ignored.

The *LBDPrevLineFIFO* control logic uses a counter, *pladvword_count[11:0]*, to counts the number of *lbd_sfu_pladvword* strobes received for the line. The *pladvword_count* counter is reset to 0 by *lbd_sfu_advline* and indicates when the number of *lbd_sfu_pladvword* strobes received is equal to *LBDNumWords*.

The *LBDPrevLineFIFO* generates a signal *plf_rdy* to indicate that it has data available. Until the first *lbd_sfu_advline* for a band has been received and after the number of *lbd_sfu_pladvword* strobes received for a line is equal to *LBDNumWords*, *plf_rdy* is always asserted. During the second and subsequent lines *plf_rdy* is deasserted whenever the *LBDPrevLineFIFO* is empty.

The last 256-bit word for a line read from DRAM can contain extra padding which should not be output to the LBD. This is because *lbd_num_words* may not fit exactly into a 256-bit DRAM word. When the count of the number of *lbd_sfu_pladvword* strobes received for a line is equal to *lbd_num_words* the *LBDPrevLineFIFO* must adjust the FIFO read address to point to the next 256-bit word boundary in the FIFO. This

can be achieved by considering the FIFO read address, *read_adr[2:0]*, will require 3 bits to address 8 locations of 64-bits. The next 256-bit aligned address is calculated by inverting the MSB of the read_adr and setting all other bits to 0.

```
if (pladvword_count == lbd_num_words) then
    read_adr[1:0] = b00
    read_adr[2] = ~read_adr[2]
```

25.8.6 LBDNextLineFIFO sub-block

Table 117. LBDNextLineFIFO Additional IO Definition

Port Name	Pins	I/O	Description
LBDNextLineFIFO Interface Signals			
nlf_rdy	1	Out	Signal indicating *LBDNextLineFIFO* is ready to be written to i.e. there is space in the FIFO.
DIU and Address Generation sub-block Signals			
nlf_diuwreq	1	Out	Signal indicating the *LBDNextLineFIFO* has 256-bits of data for writing to the DIU.
nlf_diuwack	1	In	Acknowledge from DIU that write request has been accepted and write data can be output on *nlf_diuwdata* together with *nlf_diuwvalid*.
nlf_diuwdata	1	Out	Data from *LBDNextLineFIFO* to DIU Interface. First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
nlf_diuwvalid	1	In	Signal indicating that data on *wlf_diuwdata* is valid.

25.8.6.1 General Description

The *LBDNextLineFIFO* sub-block comprises a double 256-bit buffer between the LBD and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the LBD and read by the DIU Interface and Address Generator.
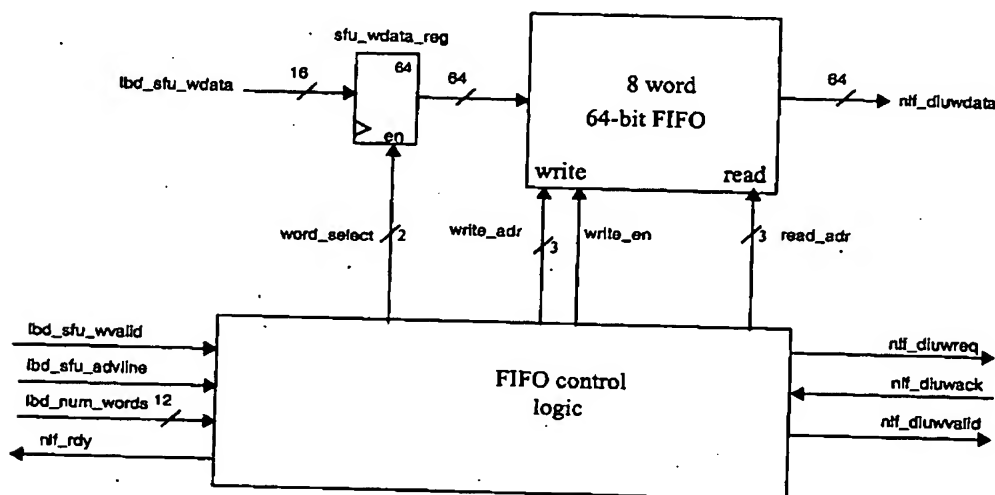
Figure 129. LBDNextLineFifo Sub-block

Whenever 4 locations in the FIFO are full the FIFO will request 256-bits of data to be written to the DIU Interface and Address Generator by asserting *nlf_diuwreq*. A signal *nlf_diuwack* indicates that the request has been accepted and *nlf_diuwreq* should be de-asserted. On receipt of *nlf_diuwack*, the data is sent to the DIU Interface as 64-bits on *nlf_diuwdata[63:0]* over 4 clock cycles. The signal *nlf_diuwvalid* indicates that the data on *nlf_diuwdata[63:0]* is valid. *nlf_diuwvalid* should be asserted with the smallest latency after *nlf_diuwack*. If the *LBDNextLineFIFO* still has 256-bits more to transfer then *nlf_diuwreq* should be asserted again.
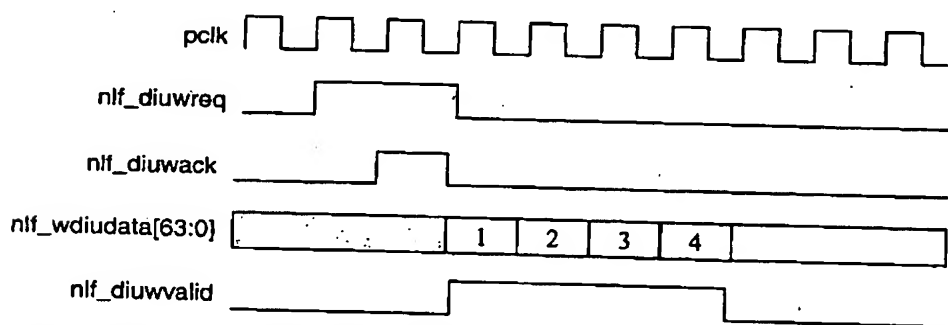
Figure 130. Timing of signals on LBDNextLineFIFO interface to DIU and Address Generator

The state diagram of the *LBDNextLineFIFO* DIU Interface is shown in Figure 131. If *sfu_go* is deasserted then the state-machine returns to its idle state.
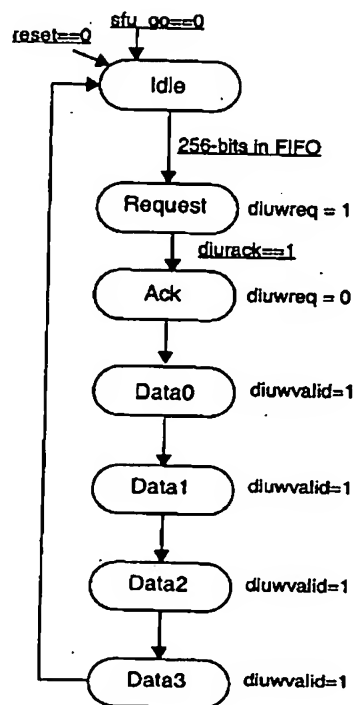
Figure 131. LBDNextLineFIFO DIU Interface State Diagram

The signal *nlf_rdy* indicates that the *LBDNextLineFIFO* has space for writing by the LBD. The LBD writes 16-bit wide data supplied on *lbd_sfu_wdata[15:0]*. *lbd_sfu_wvalid* indicates that the data is valid. The data is collected to make up a 64-bit word before being written to the FIFO.

The *LBDNextLineFIFO* control logic counts the number of *lbd_sfu_wvalid* signals. The *lbd_sfu_wvalid* counter is reset to 0 by *lbd_sfu_advline* and indicates when the number of *lbd_sfu_wvalid* strobes received is equal to *LBDNumWords*. Any data remaining in the FIFO is first flushed to DRAM with padding being added to fill a complete 256-bit word.

25.8.7   sfu_lbd_rdy Generation

The signal *sfu_lbd_rdy* is generated by ANDing *plf_rdy* from the *LBDPrevLineFIFO* and *nlf_rdy* from the *LBDNextLineFIFO*.

sfu_ldb_rdy indicates to the LBD that the SFU is available for writing i.e. there is space available in the *LBDNextLineFIFO*. After the first *lbd_sfu_advline* and before the number of *lbd_sfu_pladvword* strobes received is equivalent to the line length, *sfu_ldb_rdy* indicates that the SFU is available for both reading, i.e. there is data in the *LBDPrevLineFIFO*, and writing. Thereafter it indicates the SFU is available for writing.

25.8.8 LBD-SFU Interfaces Timing Waveform Description

In Figure 132, *sfu_wdata_reg* is the register in the SFU that registers the data written from the LBD and *virtual_nladr* is the virtual write address. Similarly, *lbd_cldata_reg* is the register in the LBD that registers the data read from the SFU and *virtual_pladr* is the virtual read address.

The main points to note from Figure 132 are:

* In clock cycle 2 *sfu_lbd_rdy* detects that it has only space to receive 2 more 16 bit words from the LBD after the current clock cycle.

* The data on *lbd_sfu_wdata* is valid and this is indicated by *lbd_sfu_wdatavalid* being asserted. Because the data is pipelined it is not captured in the SFU until clock cycle 3 (one of the two remaining spaces in the SFU FIFO).

* In clock cycle 3 *sfu_lbd_rdy* is deasserted however the LBD can not react to this signal until clock cycle 4. So in clock cycle 3 there is also valid data from the LBD which is captured in clock cycle 4 by the SFU thus taking the last available location available in the FIFO in the SFU. In clock cycle 4 the LBD has entered a pause mode and waits for sfu_lbd_rdy to be asserted again.

* *sfu_lbd_rdy* is asserted in clock cycle 7 (it could be any clock cycle), and this occurs once the SFU can has at least 2 16 bit FIFO locations available again. The LBD detects this and on clock cycle 8 it starts outputting data by asserting *lbd_sfu_wdatavalid* and putting new data out which is registered by the SFU in clock cycle 9.

There is an apparent corner case on the read side which should be highlighted. On examination this turns out to not be an issue.

Scenario 1:

sfu_lbd_rdy will go low when there is still is still 1 piece of data in the FIFO. If there is a *lbd_sfu_pladvword* pulse in the next cycle the data will appear on *sfu_lbd_pldata[15:0]*.

Scenario 2:

sfu_lbd_rdy will go low when there is still 1 piece of data in the FIFO. If there is no *lbd_sfu_pladvword* pulse in the next cycle and it is not the end of the page then the SFU will read the data for the next line from DRAM and the read FIFO will fill more, *sfu_ldb_rdy* will assert again, and so the data will appear on *sfu_lbd_pldata[15:0]*.

Scenario 3:

sfu_lbd_rdy will go low when there is still is still 1 piece of data in the FIFO. If there is no *lbd_sfu_pladvword* pulse in the next cycle and it is the end of the page then the SFU will do no more reads from DRAM, *sfu_lbd_rdy* will remain de-asserted, and the data will not be read out from the FIFO. However last line of data on the page is not needed for decoding in the LBD and will not be read by the LBD. So scenario 3 will never apply.
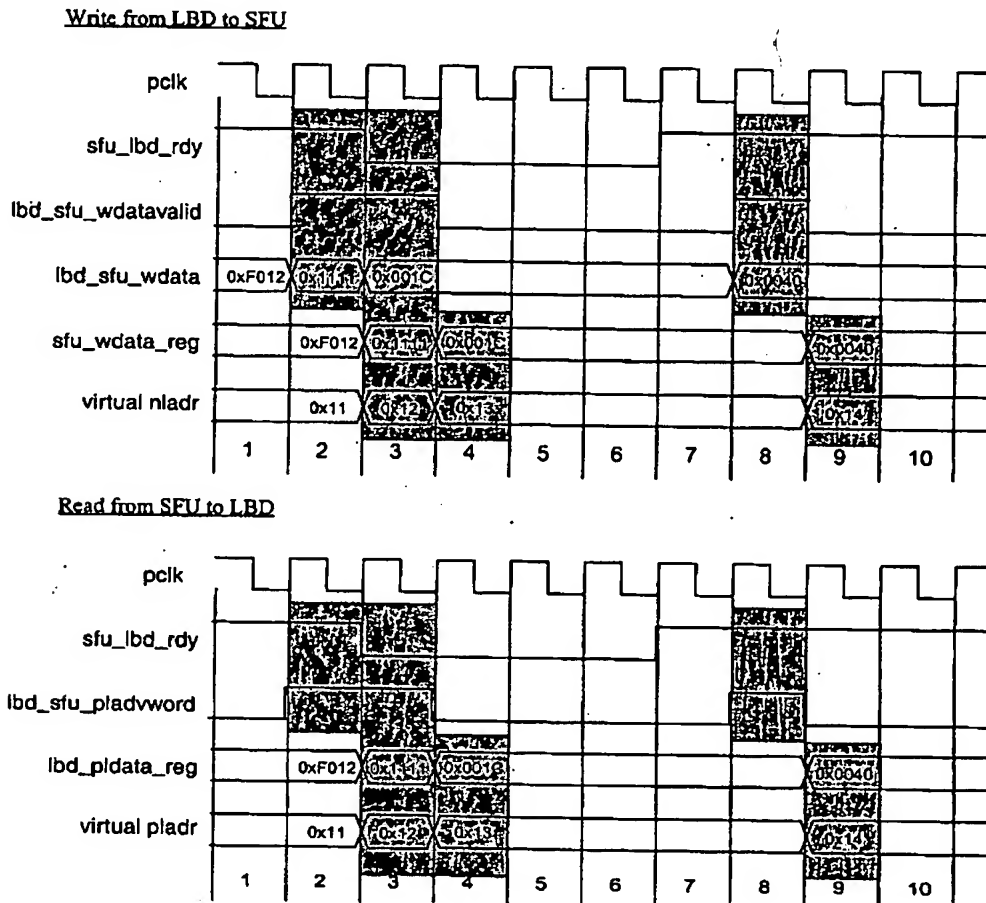
Write from LBD to SFU

Read from SFU to LBD

Figure 132. Signal waveforms between LBD and SFU

25.8.9 HCUReadLineFIFO sub-block

Table 118. HCUReadLineFIFO Additional IO Definition

Port Name	Pins	I/O	Description
DIU and Address Generation sub-block Signals			
hrf_xadvance	1	In	Signal from horizontal scaling unit 1 - supply the next dot 1 - supply the current dot
hrf_hcuendofline	1	Out	Signal lasting 1 cycle indicating then end of the HCU read line.
hrf_diurreq	1	Out	Signal indicating the *HCUReadLineFIFO* has space for 256-bits of DIU data.
hrf_diurack	1	In	Acknowledge that read request has been accepted and *hrf_diurreq* should be de-asserted.
hrf_diurdata	1	In	Data from *HCUReadLineFIFO* to DIU. First 64-bits are bits 63:0 of 256 bit word. Second 64-bits are bits 127:64 of 256 bit word. Third 64-bits are bits 191:128 of 256 bit word. Fourth 64-bits are bits 255:192 of 256 bit word.
hrf_diurvalid	1	In	Signal indicating data on *plf_diurdata* is valid.
hrf_diuidle	1	Out	Signal indicating DIU state-machine is in the IDLE state.

25.8.9.1 General Description

The *HCUReadLineFIFO* sub-block comprises a double 256-bit buffer between the HCU and the DIU Interface and Address Generator sub-block. The FIFO is implemented as 8 times 64-bit words. The FIFO is written by the DIU Interface and Address Generator sub-block and read by the HCU.
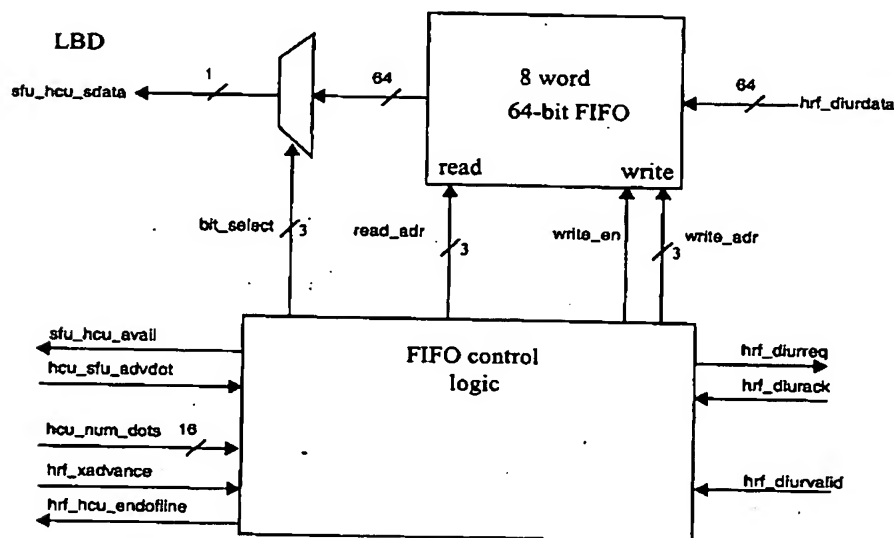
Figure 133. HCUReadLineFifo Sub-block

The DIU Interface and Address Generation (DAG) sub-block interface of the *HCUReadLineFIFO* is identical to the *LBDPrevLineFIFO* DIU interface.

Whenever 4 locations in the FIFO are free the FIFO will request 256-bits of data from the DAG sub-block by asserting *hrf_diurreq*. A signal *hrf_diurack* indicates that the request has been accepted and *hrf_diurreq* should be de-asserted.

The data is written to the FIFO as 64-bits on *hrf_diurdata[63:0]* over 4 clock cycles. The signal *hrf_diurvalid* indicates that the data returned on *hrf_diurdata[63:0]* is valid. *hrf_diurvalid* is used to generate the FIFO write enable, *write_en*, and to increment the FIFO write address, *write_adr[2:0]*. If the *HCUReadLineFIFO* still has 256-bits free then *hrf_diurreq* should be asserted again.

The *HCUReadLineFIFO* generates a signal *sfu_hcu_avail* to indicate that it has data available for the HCU. The HCU reads single-bit data supplied on *sfu_hcu_sdata*. The FIFO control logic generates a signal *bit_select* which selects the next bit of the 64-bit FIFO word to output on *sfu_hcu_sdata*. The signal *hcu_sfu_advdot* tells the *HCUReadLineFIFO* to supply the next dot (*hrf_xadvance* = 1) or the current dot (*hrf_xadvance* = 0) on *sfu_hcu_sdata* according to the *hrf_xadvance* signal from the scaling control unit in the DAG sub-block. The HCU should not generate the *hcu_sfu_advdot* signal until *sfu_hcu_avail* is true. The HCU can therefore stall waiting for the *sfu_hcu_avail* signal.

When the entire current 64-bit FIFO word has been read by the HCU *hcu_sfu_advdot* will cause the next word to be popped from the FIFO.

The last 256-bit word for a line read from DRAM and written into the *HCUReadLineFIFO* can contain dots or extra padding which should not be output to the HCU. A counter in the *HCUReadLineFIFO*, *hcuadvdot_count[15:0]*, counts the number of *hcu_sfu_advdot* strobes received from the HCU. When the count equals *hcu_num_dots[15:0]* the *HCUReadLineFIFO* must adjust the FIFO read address to point to the next 256-bit word boundary in the FIFO. This can be achieved by considering the FIFO read address, *read_adr[2:0]*, will require 3 bits to address 8 locations of 64-bits. The next 256-bit aligned address is calculated by inverting the MSB of the read_adr and setting all other bits to 0.

```
If (hcuadvdot_count == hcu_num_dots) then
    read_adr[1:0] = b00
    read_adr[2] = ~read_adr[2]
```

The DIU Interface and Address Generator sub-block scaling unit also needs to know when *hcuadvdot_count* equals *hcu_num_dots*. This condition is exported from the *HCUReadLineFIFO* as the signal *hrf_hcuendofline*. When the *hrf_hcuendofline* is asserted the scaling unit will decide based on vertical scaling whether to go back to the start of the current line or go onto the next line.

25.8.9.2 DRAM Access Limitation

The SFU must output 1 bit/cycle to the HCU. Since *HCUNumDots* may not be a multiple of 256 bits the last 256-bit DRAM word on the line can contain extra zeros. In this case, the SFU may not be able to provide 1 bit/cycle to the HCU. This could lead to a stall by the SFU. This stall could then propagate if the margins being used by the HCU are not sufficient to hide it. The maximum stall can be estimated by the calculation: DRAM service period - X scale factor * dots used from last DRAM read for HCU line.

25.8.10 DIU Interface and Address Generator Sub-block

Table 119. DIU Interface and Address Generator Additional IO Description

Port name	Pins	I/O	Description
Internal LBDPrevLineFIFO Inputs			
plf_diurreq	1	In	Signal indicating the *LBDPrevLineFIFO* has 256-bits of data free.
plf_diurack	1	Out	Acknowledge that read request has been accepted and *plf_diurreq* should be de-asserted.
plf_diurdata	1	Out	Data from the DIU to *LBDPrevLineFIFO*. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
plf_diurrvalid	1	Out	Signal indicating data on *plf_diurdata* is valid.
plf_diuidle	1	In	Signal indicating DIU state-machine is in the IDLE state.
Internal LBDNextLineFIFO Inputs			
nlf_diuwreq	1	In	Signal indicating the *LBDNextLineFIFO* has 256-bits of data for writing to the DIU.
nlf_diuwack	1	Out	Acknowledge from DIU that write request has been accepted and write data can be output on *nlf_diuwdata* together with *nlf_diuwvalid*.
nlf_diuwdata	1	In	Data from *LBDNextLineFIFO* to DIU interface. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
nlf_diuwvalid	1	In	Signal indicating that data on *wlf_diuwdata* is valid.
Internal HCUReadLineFIFO Inputs			
hrf_hcuendofline	1	In	Signal lasting 1 cycle indicating then end of the HCU read line.
hrf_xadvance	1	Out	Signal from horizontal scaling unit 1 - supply the next dot 1 - supply the current dot
hrf_diurreq	1	In	Signal indicating the *HCUReadLineFIFO* has space for 256-bits of DIU data.
hrf_diurack	1	Out	Acknowledge that read request has been accepted and *hrf_diurreq* should be de-asserted.
hrf_diurdata	1	Out	Data from *HCUReadLineFIFO* to DIU. First 64-bits are bits 63:0 of 256 bit word Second 64-bits are bits 127:64 of 256 bit word Third 64-bits are bits 191:128 of 256 bit word Fourth 64-bits are bits 255:192 of 256 bit word
hrf_diurvalid	1	Out	Signal indicating data on *plf_diurdata* is valid.
hrf_diuidle	1	In	Signal indicating DIU state-machine is in the IDLE state.

25.8.10.1 General Description

The DIU Interface and Address Generator (*DAG*) sub-block manages the bi-level buffer in DRAM. It has a DIU Write Interface for the *LBDNextLineFIFO* and a DIU Read Interface shared between the *HCUReadLineFIFO* and *LBDPrevLineFIFO*.

All DRAM address management is centralised in the *DAG*. DRAM access is pre-emptive i.e. after a FIFO unit has made an access then as soon as the FIFO has space to read or data to write a DIU access will be requested immediately. This ensures there are no unnecessary stalls introduced e.g. at the end of an LBD or HCU line.

The control logic for horizontal and vertical non-integer scaling logic is completely contained in the *DAG* sub-block. The scaling control unit exports the *hlf_xadvance* signal to the *HCUReadLineFIFO* which indicates whether to replicate the current dot or supply the next dot for horizontal scaling.

25.8.10.2 DIU Write Interface

The *LBDNextLineFIFO* generates all the DIU write interface signals directly except for *sfu_diu_wadr[21:5]* which is generated by the Address Generation logic

The DIU request from the *LBDNextLineFIFO* will be negated if its respective address pointer in DRAM is invalid i.e. *nlf_adrvalid* = 0. The implementation must ensure that no erroneous requests occur on *sfu_diu_wreq*.
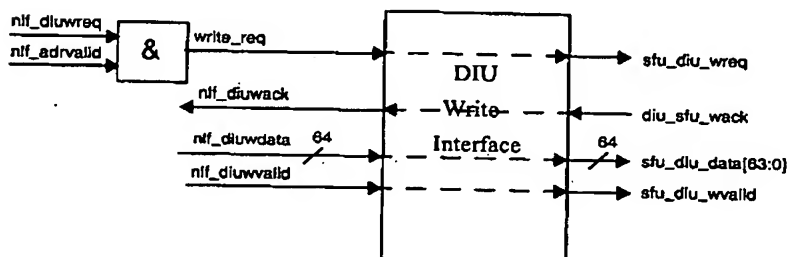
Figure 134. DIU Write Interface

25.8.10.3 DIU Read Interface

Both *HCUReadLineFIFO* and *LBDPrevLineFIFO* share the read interface. If both sources request simultaneously, then the arbitration logic implements a round-robin sharing of read accesses between the *HCUReadLineFIFO* and *LBDPrevLineFIFO*.

The DIU read request arbitration logic generates a signal, *select_hrfplf,* which indicates whether the DIU access is from the *HCUReadLineFIFO* or *LBDPrevLineFIFO* (0=*HCUReadLineFIFO*, 1 = *LBDPrevLineFIFO*). Figure 135 shows *select_hrfplf* multiplexing the returned DIU acknowledge and read data to either the *HCUReadLineFIFO* or *LBDPrevLineFIFO*.
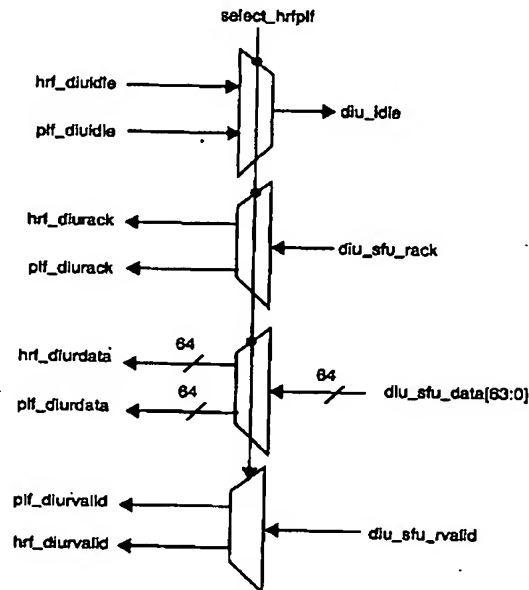
Figure 135. DIU Read Interface multiplexing by *select_hrfplf*

The DIU read request arbitration logic is shown in Figure 136. The arbitration logic will select a DIU read request on *hrf_diurreq* or *plf_diurreq* and assert *sfu_diu_rreq* which goes to the DIU. The accompanying DIU read address is generated by the Address Generation Logic. The select signal *select_hrfplf* will be set according to the arbitration winner (0=*HCUReadLineFIFO*, 1 = *LBDPrevLineFIFO*). *sfu_diu_rreq* is cleared when the DIU acknowledges the request on *diu_sfu_rack*. Arbitration cannot take place again until the DIU state-machine of the arbitration winner is in the idle state, indicated by *diu_idle*. This is necessary to ensure that the DIU read data is multiplexed back to the FIFO that requested it.
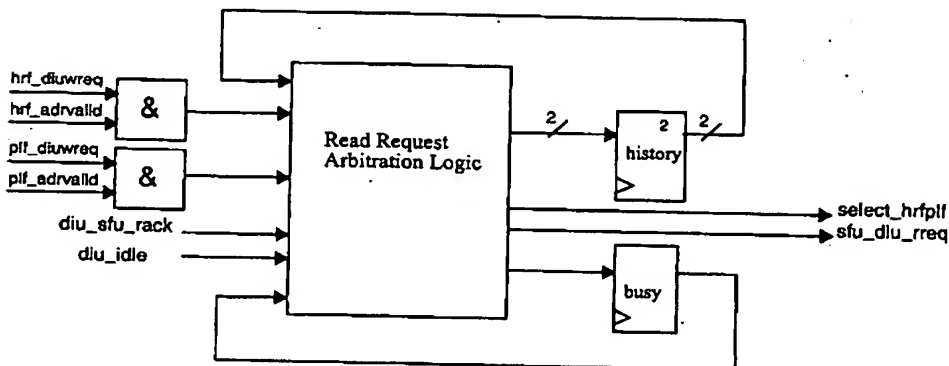
Figure 136. DIU read request arbitration logic

The DIU read requests from the *HCUReadLineFIFO* and *LBDPrevLineFIFO* will be negated if their respective addresses in DRAM are invalid, *hrf_adrvalid* = 0 or *plf_adrvalid* = 0. The implementation must ensure that no erroneous requests occur on *sfu_diu_rreq*.

If the *HCUReadLineFIFO* and *LBDPrevLineFIFO* request simultaneously, then if the request is not following immediately another DIU read port access, the arbitration logic will choose the *HCUReadLine-FIFO* by default. If there are back to back requests to the DIU read port then the arbitration logic implements a round-robin sharing of read accesses between the *HCUReadLineFIFO* and *LBDPrevLine-FIFO*.

A pseudo-code description of the DIU read arbitration is given below.

```
// history is of type (none, hrf, plf), hrf is HCUReadLineFIFO, plf is LBDPrevLineFIFO
// initialisation on reset
select_hrfplf = 0 // default choose hrf .
history = none // no DIU read access immediately preceding

// state-machine is busy between asserting sfu_diu_rreq and diu_idle = 1
// if DIU read requester state-machine is in idle state then de-assert busy
if (diu_idle == 1) then
   busy = 0

//if acknowledge received from DIU then de-assert DIU request
if (diu_sfu_rack == 1) then
   //de-assert request in response to acknowledge
   sfu_diu_rreq = 0

// if not busy then arbitrate between incoming requests
// if request detected then assert busy
if (busy == 0) then
   //if there is no request
   if (hrf_diurreq == 0) AND (plf_diurreq == 0) then
      sfu_diu_rreq = 0
      history = none
   // else there is a request
   else (
      // assert busy and request DIU read access
      busy = 1
      sfu_diu_rreq = 1
      // arbitrate in round-robin fashion between the requestors
      // if only HCUReadLineFIFO requesting choose HCUReadLineFIFO
      if (hrf_diurreq == 1) AND (plf_diurreq == 0) then
         history = hrf
         select_hrfplf = 0
      //.if only LBDPrevLineFIFO requesting choose LBDPrevLineFIFO
      if (hrf_diurreq == 0) AND (plf_diurreq == 1) then
         history = plf
         select_hrfplf = 1
      //if both HCUReadLineFIFO and LBDPrevLineFIFO requesting
      if (hrf_diurreq == 1) AND (plf_diurreq == 1) then
         // no immediately preceding request choose HCUReadLineFIFO
         if (history == none) then
            history = .hrf
            select_hrfplf = 0
         // if previous winner was HCUReadLineFIFO choose LBDPrevLineFIFO
         elsif (history == hrf) then
            history = plf
            select_hrfplf = 1
         // if previous winner was LBDPrevLineFIFO choose HCUReadLineFIFO
         elsif (history == plf) then
            history = hrf
```

```
        select_hrfplf = 0
    // end there is a request
    )
```

25.8.10.4   Address Generation Logic

The DIU interface generates the DRAM addresses of data read and written by the SFU's FIFOs.

A write request from the *LBDNextLineFIFO* on *nlf_diuwreq* causes a write request from the DIU Write Interface. The Address Generator supplies the DRAM write address on *sfu_diu_wadr[21:5]*.
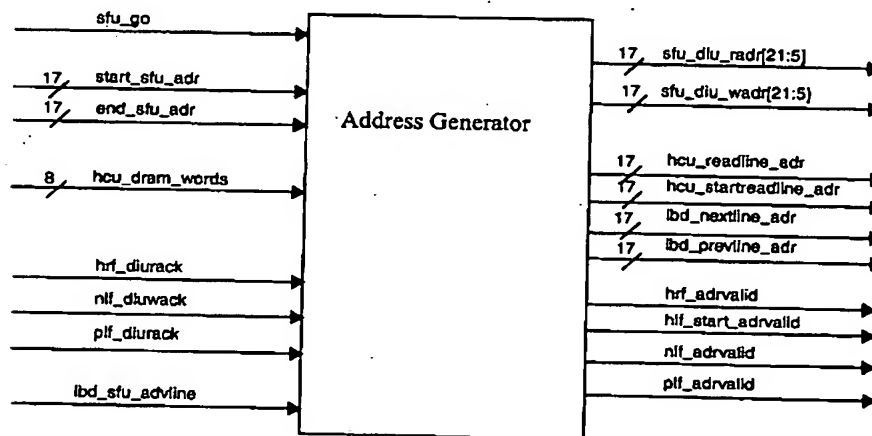
A winning read request from the DIU read request arbitration logic causes a read request from the DIU Read Interface. The Address Generator supplies the DRAM read address on *sfu_diu_radr[21:5]*.

Figure 137. Address Generation

The address generator is configured with the number of DRAM words to read in a HCU line, *hcu_dram_words[7:0]*, the first DRAM address of the SFU area, *start_sfu_adr[21:5]*, and the last DRAM address of the SFU area, *end_sfu_adr[21:5]*.

Address Generation

There are four address pointers used to manage the bi-level DRAM buffer:

   a. *hcu_readline_adr[21:5]* is the read address in DRAM for the *HCUReadLineFIFO*.

   b. *hcu_startreadline_adr[21:5]* is the start address in DRAM for the current line being read by the *HCUReadLineFIFO*.

   c. *lbd_nextline_adr[21:5]* is the write address in DRAM for the *LBDNextLineFIFO*.

   d. *lbd_prevline_adr[21:5]* is the read address in DRAM for the *LBDPrevLineFIFO*.

The current value of these address pointers are readable by the CPU.

Four corresponding address valid flags are required to indicate whether the address pointers are valid:

   a. *hlf_adrvalid*.

   b. *hlf_start_adrvalid*.

   c. *nlf_adrvalid*.

   d. *plf_adrvalid*.

DRAM requests from the FIFOs will not be issued to the DIU until the appropriate address flag is valid.

Once a request has been acknowledged, the address generation logic can calculate the address of the next 256-bit word in DRAM, ready for the next request.

Rules for address pointers

The address pointers must obey certain rules which indicate whether they are valid:

   a. $hcu\_readline\_adr[21:5]$ is only valid if it is reading earlier in the line than $lbd\_nextline\_adr[21:5]$ is writing i.e. $hlf\_adrvalid = hcu\_readline\_adr[21:5] \,!= lbd\_nextline\_adr[21:5]$.

   b. The SFU cannot overwrite the current line that the HCU is reading from i.e. $hlf\_startadrvalid = lbd\_nextline\_adr[21:5] \,!= hcu\_startreadline\_adr[21:5]$.

   c. The *LBDNextLineFIFO* must be writing earlier in the line than *LBDPrevLineFIFO* is reading and must not overwrite the current line that the HCU is reading from i.e. $nlf\_adrvalid = lbd\_nextline\_adr[21:5] \,!= lbd\_prevline\_adr[21:5]$ AND $hcu\_startreadline\_valid$.

   d. The *LBDPrevLineFIFO* can read right up to the address that *LBDNextLineFIFO* is writing i.e. $plf\_adrvalid = lbd\_prevline\_adr[21:5] \,!= lbd\_nextline\_adr[21:5]$.

   e. At startup i.e. when sfu_go is asserted, the pointers are reset to $start\_sfu\_adr[21:5]$. The first LBD *NextLineFIFO* data is allowed to be written to $lbd\_nextline\_adr[21:5]$ even though $nlf\_adrvalid$ is initially invalid.

   f. The address pointers can wrap around the SFU bi-level store area in DRAM.

X scaling of data for HCUReadLineFIFO

The signal hcu_sfu_advdot tells the *HCUReadLineFIFO* to supply the next dot or the current dot on sfu_hcu_sdata according to the $hrf\_xadvance$ signal from the scaling control unit. When $hrf\_xadvance$ is 1 the *HCUReadLineFIFO* should supply the next dot. When $hrf\_xadvance$ is 0 the *HCUReadLineFIFO* should supply the current dot.
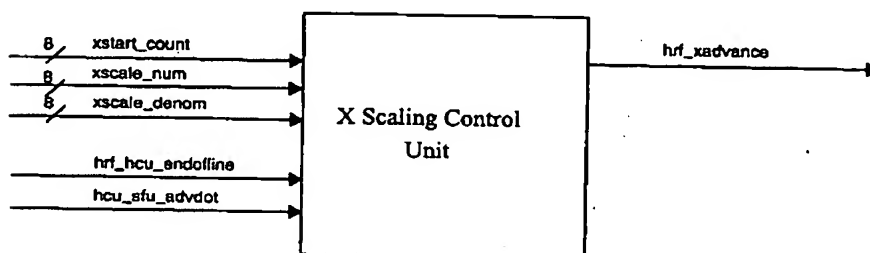
Figure 138. X scaling control unit

The algorithm for non-integer scaling is described in the pseudocode below. Note, x_scale_count should be loaded with x_start_count after reset and at the end of each line. The end of the line is indicated by $hrf\_hcuendofline$ from the *HCUReadLineFIFO*.

```
if (hcu_sfu_dotadv == 1) then
    if (x_scale_count + x_scale_denom - x_scale_num >= 0) then
        x_scale_count = x_scale_count + x_scale_denom - x_scale_num
        hrf_xadvance = 1
    else
        x_scale_count = x_scale_count + x_scale_denom
        hrf_xadvance = 0
else
```

```
        x_scale_count = x_scale_count
        hrf_xadvance = 0
```

Y scaling of data for HCUReadLineFIFO

The *HCUReadLineFIFO* counts the number of *hcu_sfu_advdot* strobes received from the HCU. When the count equals *hcu_num_dots* the *HCUReadLineFIFO* will assert *hrf_hcuendofline* for a cycle.

The algorithm for non-integer scaling is described in the pseudocode below. Note, *y_scale_count* should be loaded with *y_scale_denom* after reset.

```
if (hrf_hcu_endofline == 1) then
    if (y_scale_count + y_scale_denom - y_scale_num >= 0) then
        y_scale_count = y_scale_count + y_scale_denom - y_scale_num
        hrf_yadvance = 1
    else
        y_scale_count = y_scale_count + y_scale_denom
        hrf_yadvance = 0
else
    y_scale_count = y_scale_count
    hrf_yadvance = 0
```
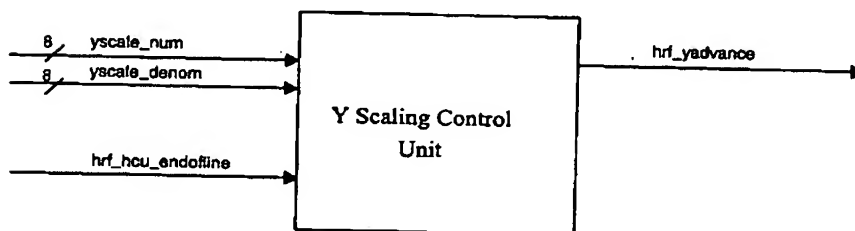

Figure 139. Y scaling control unit

When the *hrf_hcuendofline* is asserted the Y scaling unit will decide whether to go back to the start of the current line, by setting hrf_yadvance = 0, or go onto the next line, by setting hrf_yadvance = 1.

```
//if end of HCU line and advance to next line
if (hrf_hcu_endofline == 1) AND (hrf_yadvance == 1) then {
    //advance to start of next HCU line in DRAM
    hcu_startreadline_adr = hcu_startreadline_adr + hcu_dram_words
    //allow for address wraparound
    offset = hcu_startreadline_adr - end_sfu_adr
    if (offset >= 0) then
        hcu_startreadline_adr = start_sfu_adr + offset
    }
    hcu_readline_adr = hcu_startreadline_adr
    // if end of HCU line and return to start of current line
elsif (hrf_hcu_endofline == 1) AND (hrf_yadvance == 0) then
    hcu_readline_adr = hcu_startreadline_adr
```

Figure 140 shows an overview of X and Y scaling for HCU data.
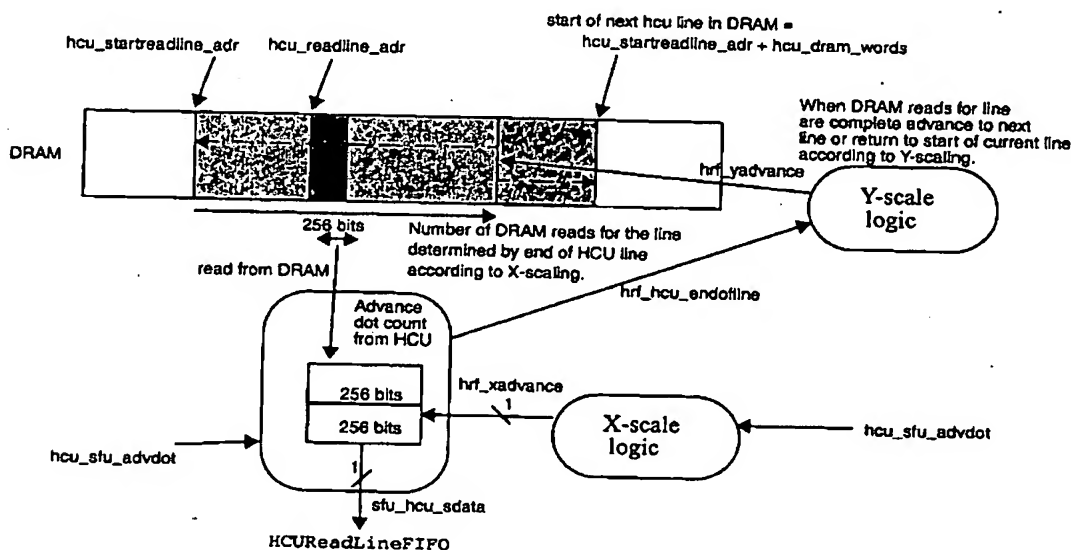
Figure 140. Overview of X and Y scaling at HCU interface

Address generator pseudo-code:

Initialization:
```
if (sfu_go rising edge) then
   //set flag to allow first write
   init = 1
   //initialise address pointers to start of SFU address space
   lbd_prevline_adr[21:5] = start_sfu_adr[21:5]
   lbd_nextline_adr[21:5] = start_sfu_adr[21:5]
   hcu_readline_adr[21:5] = start_sfu_adr[21:5]
   hcu_startreadline_adr[21:5] = start_sfu_adr[21:5]
//if first write complete
elsif (plf_adrvalid == 1) then
   // reset flag allowing first write
   init = 0
```

Address valid signals:
```
hrf_adrvalid = hcu_readline_adr != lbd_nextline_adr
hrf_startadrvalid = lbd_nextline_adr != hcu_startreadline_adr
nlf_adrvalid = init OR ((lbd_nextline_adr != lbd_prevline_adr) AND hrf_startadrvalid)
plf_adrvalid = lbd_prevline_adr != lbd_nextline_adr
```

Address pointer updating:
```
//LBDNextLineFIFO
//if DIU write acknowledge and LBDNextLineFIFO address is valid
if (diu_sfu_wack == 1 AND nlf_adrvalid) then
   //if end of SFU address range
   if (lbd_nextline_adr == end_sfu_adr) then
      //go to start of SFU address range
      lbd_nextline_adr = start_sfu_adr
   else
      //increment address pointer
```

```
            lbd_nextline_adr = lbd_nextline_adr + 1

// LBDPrevLineFIFO
//if DIU read acknowledge and LBDPrevLineFIFO address is valid
if (diu_sfu_rack == 1 AND select_hrfplf == 1 AND plf_adrvalid==1) then
   if (lbd_prevline_adr == end_sfu_adr) then
      lbd_prevline_adr = start_sfu_adr
   else
      lbd_prevline_adr = lbd_prevline_adr + 1

// HCUReadLineFIFO
//if DIU read acknowledge and HCUReadLineFIFO address is valid
if (diu_sfu_rack == 1 AND select_hrfplf == 0 AND hrf_adrvalid==1) then
   //if end of HCU line and advance to next line
   if (hrf_hcu_endofline == 1) AND (hrf_yadvance == 1) then {
      //advance to start of next HCU line in DRAM
      hcu_startreadline_adr = hcu_startreadline_adr + hcu_dram_words
      //allow for address wraparound
      offset = hcu_startreadline_adr - end_sfu_adr
      if (offset >= 0) then
         hcu_startreadline_adr = start_sfu_adr + offset
   }
   hcu_readline_adr = hcu_startreadline_adr
   //if end of HCU line and return to start of current line
   elsif (hrf_hcu_endofline == 1) AND (hrf_yadvance == 0) then
      hcu_readline_adr = hcu_startreadline_adr
   //if pointing to end of SFU address space
   elsif (hcu_readline_adr == end_sfu_adr) then
      //go to start of SFU address space
      hcu_readline_adr = start_sfu_adr
   else
      //increment address pointer
      hcu_readline_adr = hcu_readline_adr + 1
```

26 Tag Encoder (TE)

26.1 OVERVIEW

The Tag Encoder (TE) provides functionality for Netpage-enabled applications, and typically requires the presence of IR ink (although K ink can be used for tags in limited circumstances).

The TE encodes fixed data for the page being printed, together with specific tag data values into an error-correctable encoded tag which is subsequently printed in infrared or black ink on the page. The TE places tags on a triangular grid, and can be programmed for both landscape and portrait orientations.

Basic tag structures are normally rendered at 1600 dpi, while tag data is encoded into an arbitrary number of printed dots. The TE supports integer scaling in the Y-direction while the TFU supports integer scaling in the X-direction. Thus, the TE can render tags at resolutions less than 1600 dpi which can be subsequently scaled up to 1600 dpi.

The output from the TE is buffered in the Tag FIFO Unit (TFU) which is in turn used as input by the HCU. In addition, a *te_finishedband* signal is output to the end of band unit once the input tag data has been loaded from DRAM. The high level data path is shown by the block diagram in Figure 141.
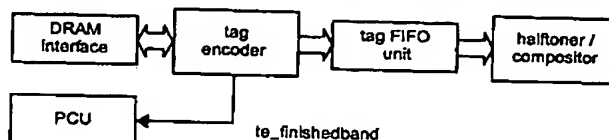
Figure 141. High level block diagram of TE in context

After passing through the HCU, the tag plane is subsequently printed with an infrared-absorptive ink that can be read by a Netpage sensing device. Since black ink can be IR absorptive, limited functionality can be provided on offset-printed pages using black ink on otherwise blank areas of the page - for example to encode buttons. Alternatively an invisible infrared ink can be used to print the position tags over the top of a regular page. However, if invisible IR ink is used, care must be taken to ensure that any other printed information on the page is printed in infrared-*transparent* CMY ink, as black ink will obscure the infrared tags. The monochromatic scheme was chosen to maximize dynamic range in blurry reading environments.

When multiple SoPEC chips are used for printing the same side of a page, it is possible that a single tag will be produced by two SoPEC chips. This implies that the TE must be able to print partial tags.

The throughput requirement for the SoPEC TE is to produce tags at half the rate of the PEC1 TE. Since the TE is reused from PEC1, the SoPEC TE over-produces by a factor of 2.

In PEC1, in order to keep up with the HCU which processes 2 dots per cycle, the tag data interface has been designed to be capable of encoding a tag in 63 cycles. This is actually accomplished in approximately 52 cycles within PEC1. If the SoPEC TE were to be modified from two dots production per cycle to a nominal one dot per cycle it should not lose the 63/52 cycle performance edge attained in the PEC1 TE.

26.2 WHAT ARE TAGS?

The first barcode was described in the late 1940's by Woodland and Silver, and finally patented in 1952 (US Patent 2,612,994) when electronic parts were scarce and very expensive. Now however, with the advent of cheap and readily available computer technology, nearly every item purchased from a shop contains a barcode of some description on the packaging. From books to CDs, to grocery items, the barcode provides a convenient way of identifying an object by a product number. The exact interpretation of the product number depends on the type of barcode. Warehouse inventory tracking systems let users define their own product number ranges, while inventory in shops must be more universally encoded so that products from one company don't overlap with products from another company. Universal Product Codes (UPC) were introduced in the mid 1970's at the request of the National Association of Food Chains for this very reason.

Barcodes themselves have been specified in a large number of formats. The older barcode formats contain characters that are displayed in the form of lines. The combination of black and white lines describe the information the barcodes contains. Often there are two types of lines to form the complete barcode: the characters (the information itself) and lines to separate blocks for better optical recognition. While the information may change from barcode to barcode, the lines to separate blocks stays constant. The lines to separate blocks can therefore be thought of as part of the constant structural components of the barcode.

Barcodes are read with specialized reading devices that then pass the extracted data onto the computer for further processing. For example, a point-of-sale scanning device allows the sales assistant to add the scanned item to the current sale, places the name of the item and the price on a display device for verification etc. Light-pens, gun readers, scanners, slot readers, and cameras are among the many devices used to read the barcodes.

To help ensure that the data extracted was read correctly, checksums were introduced as a crude form of error detection. More recent barcode formats, such as the Aztec 2D barcode developed by Andy Longacre in 1995 (US patent number US5591956), but now released to the public domain, use redundancy encoding schemes such as Reed-Solomon. Reed Solomon encoding is adequately discussed in [24], [26] and [30]. The reader is advised to refer to these sources for background information. Very often the degree of redundancy encoding is user selectable.

More recently there has also been a move from the simple one dimensional barcodes (line based) to two dimensional barcodes. Instead of storing the information as a series of lines, where the data can be extracted from a single dimension, the information is encoded in two dimensions. Just as with the original barcodes, the 2D barcode contains both information and structural components for better optical recognition. Figure 142 shows an example of a QR Code (Quick Response Code), developed by Denso of Japan (US patent number US5726435). Note the barcode cell is comprised of two areas: a data area (depends on the data being stored in the barcode), and a constant position detection pattern. The constant position detection pattern is used by the reader to help locate the cell itself, then to locate the cell boundaries, to

allow the reader to determine the original orientation of the cell (orientation can be determined by the fact that there is no 4th corner pattern).
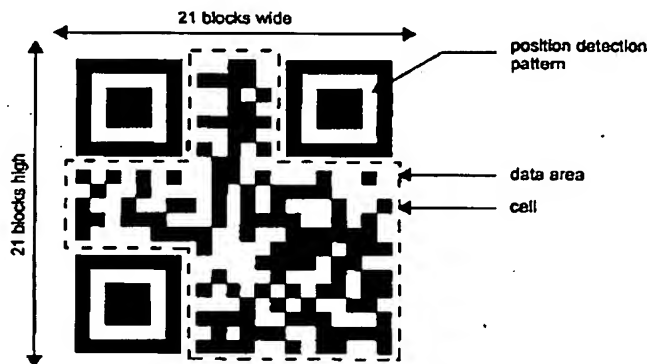

Figure 142. Example QR Code developed by Denso of Japan

The number of barcode encoding schemes grows daily. Yet very often the hardware for producing these barcodes is specific to the particular barcode format. As printers become more and more embedded, there is an increasing desire for real-time printing of these barcodes. In particular, Netpage enabled applications require the printing of 2D barcodes (or tags) over the page, preferably in infra-red ink. The tag encoder in SoPEC uses a generic barcode format encoding scheme which is particularly suited to real-time printing. Since the barcode encoding format is generic, the same rendering hardware engine can be used to produce a wide variety of barcode formats.

Unfortunately the term "barcode" is interpreted in different ways by different people. Sometimes it refers only to the data area component, and does not include the constant position detection pattern. In other cases it refers to both data and constant position detection pattern.

We therefore use the term *tag* to refer to the combination of data and any other components (such as position detection pattern, blank space etc. surround) that must be rendered to help hold or locate/read the data. A tag therefore contains the following components:

- data area(s). The data area is the whole reason that the tag exists. The tag data area(s) contains the encoded data (optionally redundancy-encoded, perhaps simply checksummed) where the bits of the data are placed within the data area at locations specified by the tag encoding scheme.
- constant background patterns, which typically includes a constant position detection pattern. These help the tag reader to locate the tag. They include components that are easy to locate and may contain orientation and perspective information in the case of 2D tags. Constant background patterns may also include such patterns as a blank area surrounding the data area or position detection pattern. These blank patterns can aid in the decoding of the data by ensuring that there is no interference between tags or data areas.

In most tag encoding schemes there is at least some constant background pattern, but it is not necessarily required by all. For example, if the tag data area is enclosed by a physical space and the reading means uses a non-optical location mechanism (e.g. physical alignment of surface to data reader) then a position detection pattern is not required.

Different tag encoding schemes have different sized tags, and have different allocation of physical tag area to constant position detection pattern and data area. For example, the QR code has 3 fixed blocks at the edges of the tag for position detection pattern (see Figure 142) and a data area in the remainder. By contrast, the Netpage tag structure (see Figures 143 and 144) contains a circular locator component, an orientation feature, and several data areas. Figure 143(a) shows the Netpage tag constant background pattern in

a resolution independent form. Figure 143(b) is the same as Figure 143(a), but with the addition of the data areas to the Netpage tag. Figure 144 is an example of dot placement and rendering to 1600 dpi for a Netpage tag. Note that in Figure 144 a single bit of data is represented by many physical output dots to form a block within the data area.
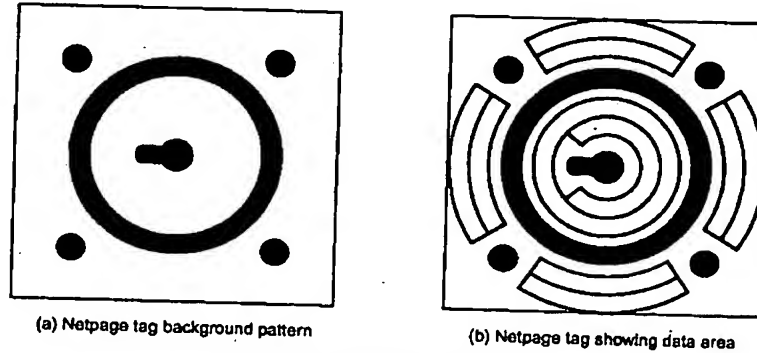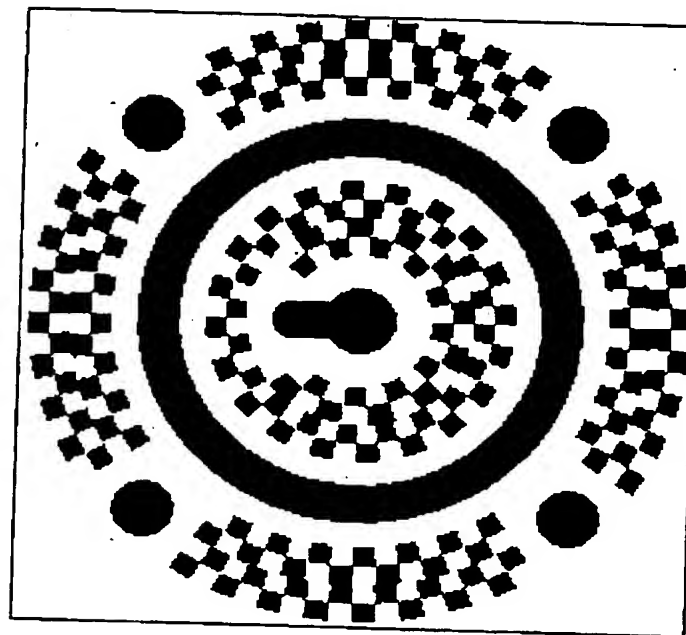

(a) Netpage tag background pattern

(b) Netpage tag showing data area

Figure 143. Netpage tag structure

Figure 144. Netpage tag with data rendered at 1600 dpi (magnified view)

26.2.1   Contents of the data area

The data area contains the data for the tag.

Depending on the tag's encoding format, a single bit of data may be represented by a number of physical printed dots. The exact number of dots will depend on the output resolution and the target reading/scan-

ning resolution. For example, in the QR code (see Figure 142), a single bit is represented by a dark module or a light module, where the exact number of dots in the dark module or light module depends on the rendering resolution and target reading/scanning resolution. For example, a dark module may be represented by a square block of printed dots (all on for binary 1, or all off for binary 0), as shown in Figure 145.
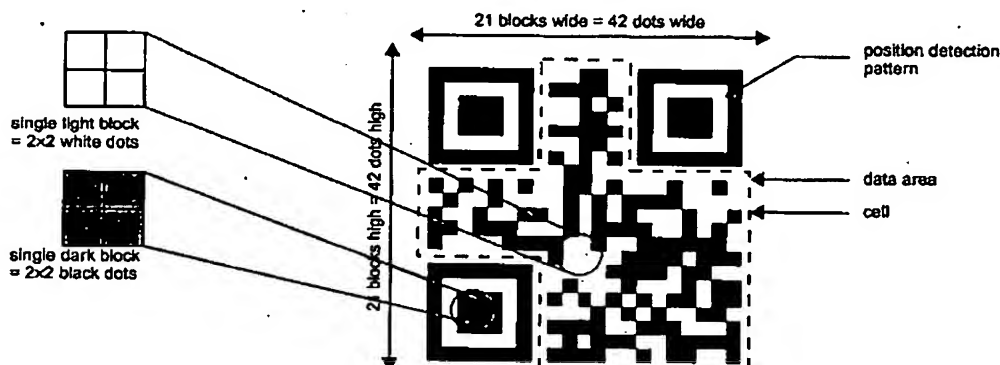

Figure 145. Example of 2×2 dots for each block of QR code

The point to note here is that a single bit of data may be represented in the printed tag by an arbitrary printed shape. The smallest shape is a single printed dot, while the largest shape is theoretically the whole tag itself, for example a giant *macrodot* comprised of many printed dots in both dimensions.

An ideal generic tag definition structure allows the generation of an arbitrary printed shape from each bit of data.

26.2.2 What do the bits represent?

Given an original number of bits of data, and the desire to place those bits into a printed tag for subsequent retrieval via a reading/scanning mechanism, the original number of bits can either be placed directly into the tag, or they can be redundancy-encoded in some way. The exact form of redundancy encoding will depend on the tag format.

The placement of data bits within the data area of the tag is directly related to the redundancy mechanism employed in the encoding scheme. The idea is generally to place data bits together in 2D so that burst errors are averaged out over the tag data, thus typically being correctable. For example, all the bits of Reed-Solomon codeword would be spread out over the entire tag data area so to minimize being affected by a burst error.

Since the data encoding scheme and shape and size of the tag data area are closely linked, it is desirable to have a generic tag format structure. This allows the same data structure and rendering embodiment to be used to render a variety of tag formats.

26.2.2.1 Fixed and variable data components

In many cases, the tag data can be reasonably divided into fixed and variable components. For example, if a tag holds N bits of data, some of these bits may be fixed for all tags while some may vary from tag to tag.

For example, the Universal product code allows a country code and a company code. Since these bits don't change from tag to tag, these bits can be defined as fixed, and don't need to be provided to the tag encoder each time, thereby reducing the bandwidth when producing many tags.

Another example is Netpage tags. A single printed page contains a number of Netpage tags. The page-id will be constant across all the tags, even though the remainder of the data within each tag may be different for each tag. By reducing the amount of variable data being passed to SoPEC's tag encoder for each tag, the overall bandwidth can be reduced.

Depending on the embodiment of the tag encoder, these parameters will be either implicit or explicit, and may limit the size of tags renderable by the system. For example, a software tag encoder may be completely variable, while a hardware tag encoder such as SoPEC's tag encoder may have a maximum number of tag data bits.

26.2.2.2 Redundancy-encode the tag data within the tag encoder

Instead of accepting the complete number of TagData bits encoded by an external encoder, the tag encoder accepts the basic non-redundancy-encoded data bits and encodes them as required for each tag. This leads to significant savings of bandwidth and on-chip storage.

In SoPEC's case for Netpage tags, only 120 bits of original data are provided per tag, and the tag encoder encodes these 120 bits into 360 bits. By having the redundancy encoder on board the tag encoder the effective bandwidth and internal storage required is reduced to only 33% of what would be required if the encoded data was read directly.

26.3   PLACEMENT OF TAGS ON A PAGE

The TE places tags on the page in a triangular grid arrangement as shown in Figure 146.
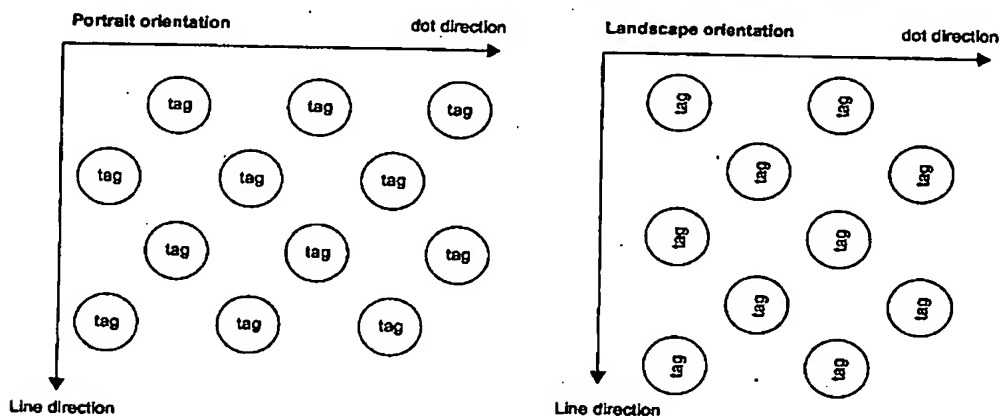

Figure 146. Placement of tags for portrait & landscape printing

The triangular mesh of tags combined with the restriction of no overlap of columns or rows of tags means that the process of tag placement is greatly simplified. For a given line of dots, all the tags on that line correspond to the same part of the general tag structure. The triangular placement can be considered as alternative lines of tags, where one line of tags is inset by one amount in the dot dimension, and the other line of dots is inset by a different amount. The dot inter-tag gap is the same in both lines of tag, and is different from the line inter-tag gap.

Note also that as long as the tags themselves can be rotated, portrait and landscape printing are essentially the same - the placement parameters of line and dot are swapped, but the placement mechanism is the same.

The general case for placement of tags therefore relies on a number of parameters, as shown in Figure 147.
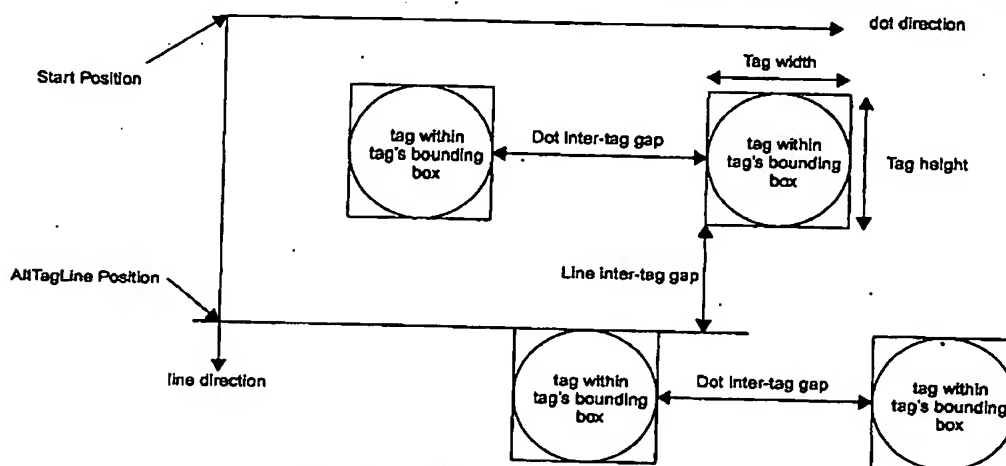

Figure 147. General representation of tag placement

The parameters are more formally described in Table 120. Note that these are placement parameters and not registers.

Table 120. Tag placement parameters

Parameter	description	restrictions
Tag height	The number of dot lines in a tag's bounding box	minimum 1
Tag width	The number of dots in a single line of the tag's bounding box. The number of dots in the tag itself may vary depending on the shape of the tag, but the number of dots in the bounding box will be constant (by definition).	minimum 1
Dot inter-tag gap	The number of dots from the edge of one tag's bounding box to the start of the next tag's bounding box, in the dot direction.	minimum = 0
Line inter-tag gap	The number of dot lines from the edge of one tag's bounding box to the start of the next tag's bounding box, in the line direction.	minimum = 0
Start Position	Defines the status of the top left dot on the page - is an offset in dot & row within the tag or the inter-tag gap.	-
AltTagLinePosition	Defines the status for the start of the alternate row of tags. Is an offset in dot within the tag or within the dot inter-tag gap (the row position is always 0).	-

26.4 BASIC TAG ENCODING PARAMETERS

SoPEC's tag encoder imposes range restrictions on tag encoding parameters as a direct result of on-chip buffer sizes. Table 121 lists the basic encoding parameters as well as range restrictions where appropriate. Although the restrictions were chosen to take the most likely encoding scenarios into account, it is a sim-

ple matter to adjust the buffer sizes and corresponding addressing to allow arbitrary encoding parameters in future implementations.

Table 121. Encoding parameters

name	definition	maximum value imposed by TE
W	page width	2^{14} dotpairs or 20.48 inches at 1600 dpi
S	tag size	typical tag size is 2mm x 2mm maximum tag size is 384 dots x 384 dots before scaling i.e. 6 mm x 6 mm at 1600 dpi
N	number of dots in each dimension of the tag	384 dots before scaling
E	redundancy encoding for tag data	Reed-Solomon GF(2^4) at 5:10 or 7:8
D_F	size of fixed data (unencoded)	40 or 56 bits
R_F	size of redundancy-encoded fixed data	120 bits
D_V	size of variable data (unencoded)	120 or 112 bits
R_V	size of redundancy-encoded variable data	360 or 240 bits
T	tags per page width	85 packed 6mm x 6mm tags (384 x 384 dots) will fit in 20.48 inches

The fixed data for the tags on a page need only be supplied to the TE once. It can be supplied as 40 or 56 bits of unencoded data and encoded within the TE as described in Section 26.4.1. Alternatively it can be supplied as 120 bits of pre-encoded data (encoded arbitrarily).

The variable data for the tags on a page are those 112 or 120 data bits that are variable for each tag. Variable tag data is supplied as part of the band data, and is always encoded by the TE as described in Section 26.4.1, but may itself be arbitrarily pre-encoded.

26.4.1 Redundancy encoding

The mapping of data bits (both fixed and variable) to redundancy encoded bits relies heavily on the method of redundancy encoding employed. Reed-Solomon encoding was chosen for its ability to deal with burst errors and effectively detect and correct errors using a minimum of redundancy. Reed Solomon encoding is adequately discussed in [24], [26] and [30]. The reader is advised to refer to these sources for background information.

In this implementation of the TE we use Reed-Solomon encoding over the Galois Field GF(2^4). Symbol size is 4 bits. Each codeword contains 15 4-bit symbols for a codeword length of 60 bits. The primitive polynomial is $p(x) = x^4 + x + 1$, and the generator polynomial is $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^{2t})$, where $t =$ the number of symbols that can be corrected.

Of the 15 symbols, there are two possibilities for encoding:
* RS(15, 5): 5 symbols original data (20 bits), and 10 redundancy symbols (40 bits). The 10 redundancy symbols mean that we can correct up to 5 symbols in error. The generator polynomial is therefore $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^{10})$.
* RS(15, 7): 7 symbols original data (28 bits), and 8 redundancy symbols (32 bits). The 8 redundancy symbols mean that we can correct up to 4 symbols in error. The generator polynomial is $g(x) = (x+\alpha)(x+\alpha^2)...(x+\alpha^8)$.

In the first case, with 5 symbols of original data, the total amount of original data per tag is 160 bits (40 fixed, 120 variable). This is redundancy encoded to give a total amount of 480 bits (120 fixed, 360 variable) as follows:

- Each tag contains up to 40 bits of fixed original data. Therefore 2 codewords are required for the fixed data, giving a total encoded data size of 120 bits. Note that this fixed data only needs to be encoded once per page.
- Each tag contains up to 120 bits of variable original data. Therefore 6 codewords are required for the variable data, giving a total encoded data size of 360 bits.

In the second case, with 7 symbols of original data, the total amount of original data per tag is 168 bits (56 fixed, 112 variable). This is redundancy encoded to give a total amount of 360 bits (120 fixed, 240 variable) as follows:

- Each tag contains up to 56 bits of fixed original data. Therefore 2 codewords are required for the fixed data, giving a total encoded data size of 120 bits. Note that this fixed data only needs to be encoded once per page.
- Each tag contains up to 112 bits of variable original data. Therefore 4 codewords are required for the variable data, giving a total encoded data size of 240 bits.

The choice of data to redundancy ratio depends on the application.

26.5 DATA STRUCTURES USED BY TAG ENCODER

26.5.1 Tag Format Structure

The Tag Format Structure (TFS) is the template used to render tags, optimized so that the tag can be rendered in real time. The TFS contains an entry for each dot position within the tag's bounding box. Each entry specifies whether the dot is part of the constant background pattern or part of the tag's data component (both fixed and variable).

The TFS is very similar to a bitmap in that it contains one entry for each dot position of the tag's bounding box. The TFS therefore has *TagHeight* × *TagWidth* entries, where *TagHeight* matches the height of the bounding box for the tag in the line dimension, and *TagWidth* matches the width of the bounding box for the tag in the dot dimension. A single line of TFS entries for a tag is known as a *tag line structure*.

The TFS consists of *TagHeight* number of *tag line structures*, one for each 1600 dpi line in the tag's bounding box. Each tag line structure contains three contiguous tables, known as tables A, B, and C. Table A contains 384 2-bit entries, one entry for each of the maximum number of dots in a single line of a tag (see Table 121). The actual number of entries used should match the size of the bounding box for the tag in the dot dimension, but all 384 entries must be present[1]. Table B contains 32 9-bit data addresses that refer to (in order of appearance) the data dots present in the particular line. All 32 entries must be present, even if fewer are used. Table C contains two 5-bit pointers into table B, and is stored in the 10 low bits of the next 32-bit word (the upper 22 bits are unused). The total length of each tag line structure is therefore 34 × 32-bit words. Padding (18 × 32-bit words) is inserted after every 7 tag line structures to keep each tag line

---

1. This is done so that it is possible to go from one line within a tag to the next by simply adding 33 in 32-bit based addressing to DRAM.

structure completely within a 1KByte boundary (thus a TFS containing *TagHeight* tag line structures requires a *TagHeight*/7 rounded up KBytes). The structure of a TFS is shown in Figure 148.
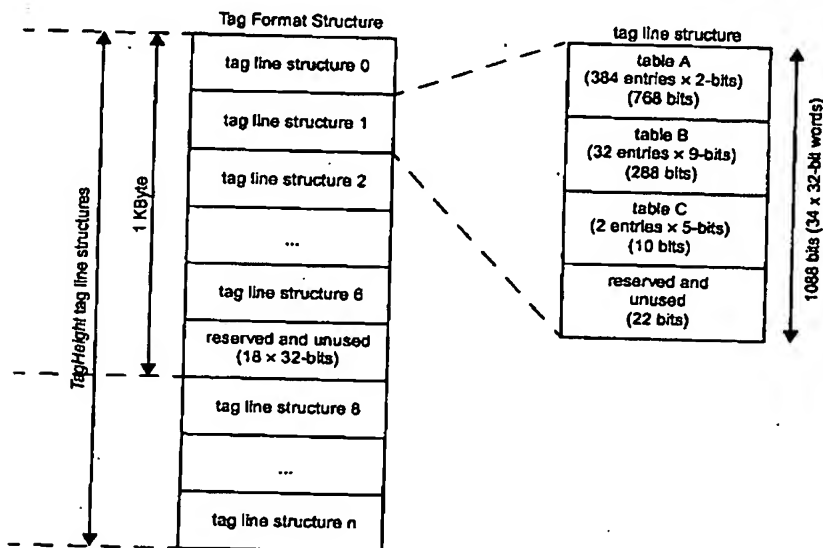

Figure 148. Composition of SoPEC's tag format structure

A full description of the interpretation and usage of Tables A, B and C is given in section 26.8.3 on page 414.

26.5.1.1 Scaling a tag

If the size of the printed dots is too small, then the tag can be scaled in one of several ways. Either the tag itself can be scaled by N dots in each dimension, which increases the number of entries in the TFS. As an alternative, the output from the TE can be scaled up by pixel replication via a scale factor greater than 1 in the both the TE and TFU.

For example, if the original TFS was 21 × 21 entries, and the scaling were a simple 2 × 2 dots for each of the original dots, we could increase the TFS to be 42 × 42. To generate the new TFS from the old, we would repeat each entry across each line of the TFS, and then we would repeat each line of the TFS. The net number of entries in the TFS would be increased fourfold (2 × 2).

The TFS allows the creation of *macrodots* instead of simple scaling. Looking at Figure 149 for a simple example of a 3 × 3 dot tag, we may want to produce a physically large printed form of the tag, where each of the original dots was represented by 7 × 7 printed dots. If we simply performed replication by 7 in each dimension of the original TFS, either by increasing the size of the TFS by 7 in each dimension or putting a scale-up on the output of the tag generator output, then we would have 9 sets of 7 × 7 square blocks.

Instead, we can replace each of the original dots in the TFS by a 7×7 dot definition of a rounded dot. Figure 150 shows the results.
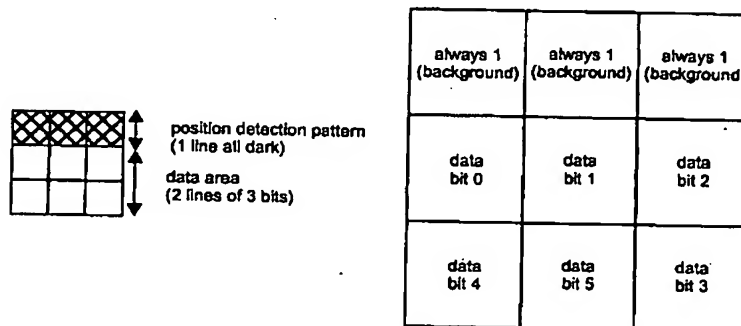
always 1 (background)	always 1 (background)	always 1 (background)
data bit 0	data bit 1	data bit 2
data bit 4	data bit 5	data bit 3

position detection pattern (1 line all dark)

data area (2 lines of 3 bits)

Figure 149. Simple 3x3 tag structure

Legend

	constant 0
	constant 1
b0	data bit 0
b1	data bit 1
b2	data bit 2
b3	data bit 3
b4	data bit 4
b5	data bit 5
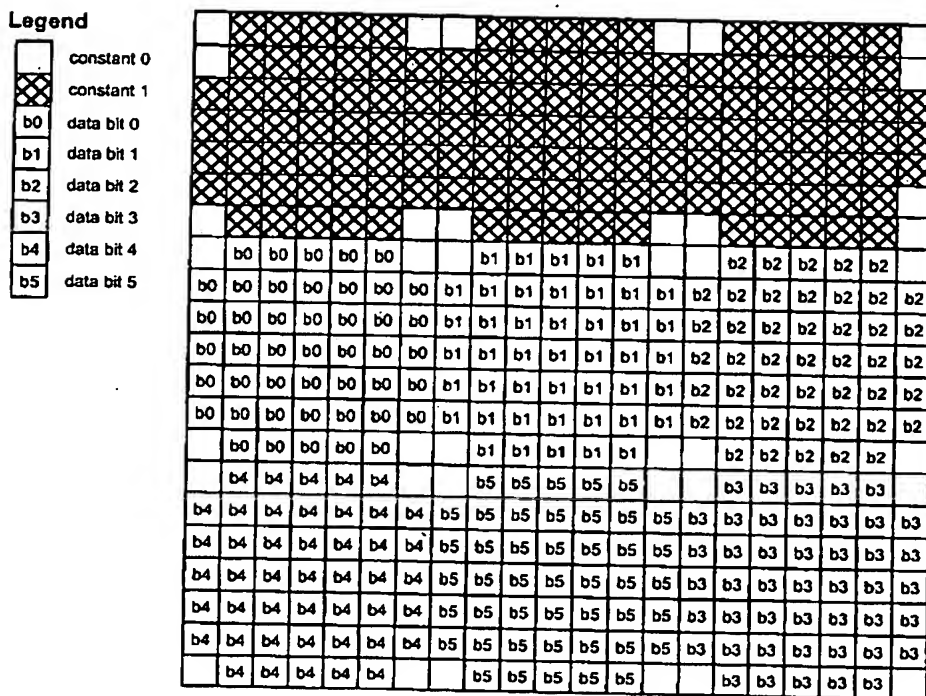
Figure 150. 3x3 tag redesigned for 21 x 21 area (not simple replication)

Consequently, the higher the resolution of the TFS the more printed dots can be printed for each *macrodot*, where a macrodot represents a single data bit of the tag. The more dots that are available to produce a macrodot, the more complex the pattern of the macrodot can be. As an example, Figure 144 on page 360 shows the Netpage tag structure rendered such that the data bits are represented by an average of 8 dots ×

8 dots (at 1600 dpi), but the actual shape structure of a dot is not square. This allows the printed Netpage tag to be subsequently read at any orientation.

26.5.2 Raw tag data

The TE requires a band of unencoded variable tag data if variable data is to be included in the tag bit-plane. A band of unencoded variable tag data is a set of contiguous unencoded tag data records, in order of encounter top left of printed band from top left to lower right.

An unencoded tag data record is 128 bits arranged as follows: bits 0-111 or 0-119 are the bits of raw tag data, bit 120 is a flag used by the TE (*TagIsPrinted*), and the remaining 7 bits are reserved (and should be 0). Having a record size of 128 bits simplifies the tag data access since the data of two tags fits into a 256-bit DRAM word. It also means that the flags can be stored apart from the tag data, thus keeping the raw tag data completely unrestricted. If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

The *TagIsPrinted* flag allows the effective specification of a tag resolution mask over the page. For each tag position the *TagIsPrinted* flag determines whether any of the tag is printed or not. This allows arbitrary placement of tags on the page. For example, tags may only be printed over particular active areas of a page. The *TagIsPrinted* flag allows only those tags to be printed. *TagIsPrinted* is a 1 bit flag with values as shown in Table 122.

Table 122. TagIsPrinted values

Value	description
0	Don't print the tag in this tag position. Output 0 for each dot within the tag bounding box.
1	Print the tag as specified by the various tag structures.

26.5.3 DRAM storage requirements

The total DRAM storage required by a single band of raw tag data depends on the number of tags present in that band. Each tag requires 128 bits. Consequently if there are N tags in the band, the size in DRAM is *16N* bytes.

The maximum size of a line of tags is 163×128 bits. When maximally packed, a row of tags contains 163 tags (see Table 121) and extends over a minimum of 126 print lines. This equates to 282 KBytes over a Letter page.

The total DRAM storage required by a single TFS is *TagHeight/7* KBytes (including padding). Since the likely maximum value for *TagHeight* is 384 (given that SoPEC restricts *TagWidth* to 384), the maximum size in DRAM for a TFS is 55 KBytes.

26.5.4 DRAM access requirements

The TE has two separate read interfaces to DRAM for raw tag data, TD, and tag format structure, TFS.

The memory usage requirements are shown in Table 123. Raw tag data is stored in the compressed page store

Table 123. Memory usage requirements

Block	Size	Description
Compressed page store	2048 Kbytes	Compressed data page store for Bi-level, contone and raw tag data.
Tag Format Structure	55 Kbyte (384 dot line tags @ 1600 dpi)	55 kB in PEC1 for 384 dot line tags (the benchmark) at 1600 dpi 2.5 mm tags (1/10th inch) @ 1600 dpi require 160 dot lines = 160/384 x55 or 23 kB 2.5 mm tags @ 800 dpi require 80/384 x55 = 12 kB

The TD interface will read 256-bits from DRAM at a time. Each 256-bit read returns 2 times 128-bit tags. The TD interface to the DIU will be a 256-bit double buffer. If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

The TFS interface will also read 256-bits from DRAM at a time. The TFS required for a line is 136 bytes. A total of 5 times 256-bit DRAM reads is required to read the TFS for a line with 192 unused bits in the fifth 256-bit word. A 136-byte double-line buffer will be implemented to store the TFS data.

The TE's DIU bandwidth requirements are summarized in Table 124.

Table 124. DRAM bandwidth requirements

Block Name	Direction	Maximum number of cycles between each 256-bit DRAM access	Peak Bandwidth (bits/cycle)	Average Bandwidth (bits/cycle)
TD	Read	Single 256 bit reads[1].	1.02	1.02
TFS	Read	Single 256 bit reads[2]. TFS is 136 bytes. This means there is unused data in the fifth 256 bit read. A total of 5 reads is required.	0.093	0.093

1: Each 2mm tag lasts 126 dot cycles and requires 128 bits. This is a rate of 256 bits every 252 cycles.

2: 17 x 64 bit reads per line in PEC1 is 5 x 256 bit reads per line in SoPEC with unused bits in the last 256-bit read.

26.5.5   Tag sizes

SoPEC allows for tags to be between 0 to 384 dots. A typical 2 mm tag requires 126 dots. Short tags do not change the internal bandwidth or throughput behaviours at all. Tag height is specified so as to allow the DRAM storage for raw tag data to be specified. Minimum tag width is a condition imposed by throughput limitations, so if the width is too small TE cannot consistently produce 2 dots per cycle across several tags (also there are raw tag data bandwidth implications). Thinner tags still work, they just take longer and/or need scaling.

26.6   IMPLEMENTATION

26.6.1   Tag Encoder Architecture

A block diagram of the TE can be seen below.

Figure 151. TE Block Diagram

The TE writes lines of bi-level tag plane data to the TFU for later reading by the HCU. The TE is responsible for merging the encoded tag data with the tag structure (interpreted from the TFS). Y-integer scaling of tags is performed in the TE with X-integer scaling of the tags performed in the TFU. The encoded tag layer is generated 2 bits at a time and output to the TFU at this rate. The HCU however only consumes 1 bit per cycle from the TFU. The TE must provide support for 126dot Tags (2mm densely packed) with 108 Tags per line with 128bits per tag.

The tag encoder consists of a TFS interface that loads and decodes TFS entries, a tag data interface that loads tag raw data, encodes it, and provides bit values on request, and a state machine to generate appropriate addressing and control signals. The TE has two separate read interfaces to DRAM for raw tag data, TD, and tag format structure, TFS.

It is possible that the raw tag data interface, the TD, to the DIU could be replaced by a hardware state machine at a later stage. This would allow flexibility in the generation of tags. Support for Y scaling needs to be added to the PEC1 TE. The PEC1 TE already allows stalling at its output during a line when *tfu_te_oktowrite* is deasserted.

26.6.2 Y-Scaling output lines

In order to support scaling in the Y direction the following modifications to the PEC1 TE are suggested to the Tag.Data Interface, Tag Format Structure Interface and TE Top Level:

- for Tag Data Interface: program the configuration registers of *Table 126*, *firstTagLineHeight* and *tagMaxLine* with true value i.e. not multiplied up by the scale factor *YScale*. Within the Tag Data interface there are two counters, *countx* and *county* that have a direct bearing on the *rawTagDataAddr* generation. *countx* decrements as tags are read from DRAM. It is reset to *NumTags[RtdTagSense]* at start of each line of tags. *county* is decremented as each line of tags is completely read from DRAM i.e. *countx* = 0. Scaling may be performed by counting the number of times *countx* reaches zero and only decrementing *county* when this number reaches *YScale*. This will cause the TagData Interface to read each line of tag data *NumTags[RtdTagSense]* * *YScale* times.

- for Tag Format Structure Interface: The implication of Y-scaling for the TFS is that each Tag Line Structure is used *YScale* times. This may be accomplished in either of two ways:
  - For each Tag Line Structure read it once from DRAM and reuse *YScale* times. This involves gating the control of TFS buffer flipping with *YScale*. Because of the way in which this *advTfsLine* and *advTagLine* related functionality is coded in the PEC1 TFS this solution is judged to be error-prone.
  - Fetch each TagLineStructure *YScale* times. This solution involves controlling the activity of *currTfsAddr* with *YScale*.
    In SoPEC the TFS must supply five addresses to the DIU to read each individual Tag Line Structure. The DIU returns 4*64-bit words for each of the 5 accesses. This is different from the behaviour in PEC1, where one address is given and 17 data-words were returned by the DIU.
    Since the behaviour of the *currTfsAddr* must be changed to meet the requirements of the SoPEC DIU it makes sense to include the Y-Scaling into this change i.e. a count of the number of completed sets of 5 accesses to the DIU is compared to YScale. Only when this count equals YScale can *currTfsAddr* be loaded with the base address of the next lines Tag Line Structure in DRAM, otherwise it is re-loaded with the base address of the current lines Tag Line Structure in DRAM.

- For Top Level: The Top Level of the TE has a counter, *LinePos*, which is used to count the number of completed output lines when in a tag gap or in a line of tags. At the start (i.e. top-left hand dot-pair) of a gap or tag *LinePos* is loaded with either *TagGapLine* or *TagMaxLine*. The value of *LinePos* is decremented at last dot-pair in line. Y-Scaling may be accomplished by gating the decrement of *LinePos* based on *YScale* value

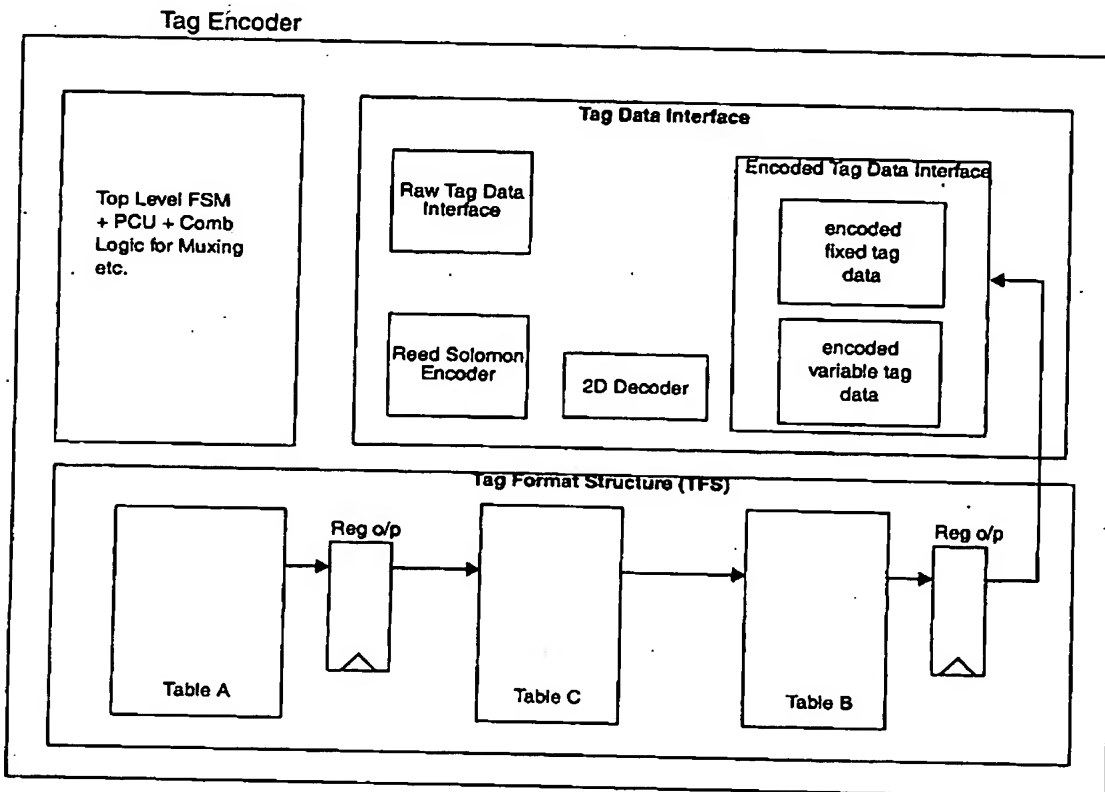### 26.6.3   TE Physical Hierarchy

Tag Encoder

Figure 152. TE Hierarchy

Figure 152 above illustrates the structural hierarchy of the TE. The top level contains the Tag Data Interface (TDI), Tag Format Structure (TFS), and an FSM to control the generation of dot pairs along with a clocked process to carry out the PCU read/write decoding. There is also some additional logic for muxing the output data and generating other control signals.

At the highest level, the TE state machine processes the output lines of a page one line at a time, with the starting position either in an inter-tag gap or in a tag (a SoPEC may be only printing part of a tag due to multiple SoPECs printing a single line).

If the current position is within an inter-tag gap, an output of 0 is generated. If the current position is within a tag, the tag format structure is used to determine the value of the output dot, using the appropriate encoded data bit from the fixed or variable data buffers as necessary. The TE then advances along the line of dots, moving through tags and inter-tag gaps according to the tag placement parameters.

26.6.4 IO Definitions

Table 125. TE Port List

Port Name	Pits	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
Bandstore Signals			
cdu_endofbandstore[21:5]	17	In	Address of the end of the current band of data. 256-bit word aligned DRAM address.
cdu_startofbandstore[21:5]	17	In	Address of the start of the current band of data. 256-bit word aligned DRAM address.
te_finishedband	1	Out	TE finished band signal to PCU and ICU.
PCU Interface data and control signals			
pcu_addr[8:2]	7	In	PCU address bus. 7 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
te_pcu_datain[31:0]	32	Out	Read data bus from the TE to the PCU.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_te_sel	1	In	Block select from the PCU. When *pcu_te_sel* is high both *pcu_addr* and *pcu_dataout* are valid.
te_pcu_rdy	1	Out	Ready signal to the PCU. When *te_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *te_pcu_datain* is valid.
TD (raw Tag Data) DIU Read Interface signals			
td_diu_rreq	1	Out	TD requests DRAM read. A read request must be accompanied by a valid read address.
td_diu_radr[21:5]	17	Out	TD read address to DIU. 17 bits wide (256-bit aligned word).
diu_td_rack	1	In	Acknowledge from DIU that TD read request has been accepted and new read address can be placed on *te_diu_radr*.
diu_data[63:0]	64	In	Data from DIU to TE. First 64-bits are bits 63:0 of 256 bit word; Second 64-bits are bits 127:64 of 256 bit word; Third 64-bits are bits 191:128 of 256 bit word; Fourth 64-bits are bits 255:192 of 256 bit word.
diu_td_rvalid	1	In	Signal from DIU telling TD that valid read data is on the *diu_data* bus.
TFS (Tag Format Structure) DIU Read Interface signals			
tfs_diu_rreq	1	Out	TFS requests DRAM read. A read request must be accompanied by a valid read address.
tfs_diu_radr[21:5]	17	Out	TFS Read address to DIU 17 bits wide (256-bit aligned word).
diu_tfs_rack	1	In	Acknowledge from DIU that TFS read request has been accepted and new read address can be placed on *tfs_diu_radr*.

Table 125. TE Port List

Port Names	Pins	I/O	Description
diu_data[63:0]	64	In	Data from DIU to TE. First 64-bits are bits 63:0 of 256 bit word; Second 64-bits are bits 127:64 of 256 bit word; Third 64-bits are bits 191:128 of 256 bit word; Fourth 64-bits are bits 255:192 of 256 bit word.
diu_tfs_rvalid	1	In	Signal from DIU telling TFS that valid read data is on the *diu_data* bus.
TFU Interface data and control signals			
tfu_te_oktowrite	1	In	Ready signal indicating TFU has space available and is ready to be written to. Also asserted from the point that the TFU has recieved its expected number of bytes for a line until the next *te_tfu_wradvline*
te_tfu_wdata[7:0]	8	Out	Write data for TFU.
te_tfu_wdatavalid	1	Out	Write data valid signal. This signal remains high whenever there is valid output data on *te_tfu_wdata*
te_tfu_wradvline	1	Out	Advance line signal strobed when the last byte in a line is placed on *te_tfu_wdata*

26.6.5 Configuration Registers

The configuration registers in the TE are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for the description of the protocol and timing diagrams for reading and writing registers in the TE. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes the lower 2 bits of the PCU address bus are not required to decode the address space for the TE. Table 126 lists the configuration registers in the TE.

Registers which address DRAM are 64-bit DRAM word aligned as this is the case for the PEC1 TE. SoPEC assumes a 256-bit DRAM word size. If the TE can be easily modified then the DRAM word addressing should be modified to 256-bit word aligned addressing. Otherwise, software should program these the 64-bit word aligned addresses on a 256-bit DRAM word boundary..

Table 126. TE Configuration Registers

Address (TE_base+)	register name	#bits	value on reset	description
Control registers				
0x00	Reset	1	1	A write to this register causes a reset of the TE. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress
0x04	Go	1	0	Writing 1 to this register starts the TE. Writing 0 to this register halts the TE. When Go is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When Go is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). NextBandEnable is cleared when Go is asserted. The TFU must be started before the TE is started. This register can be read to determine if the TE is running (1 = running, 0 = stopped).

Table 126. TE Configuration Registers

Address TE base+	register name	bits	value on reset	description
Setup registers (constant for processing of a page)				
0x40	TfsStartAdr (64-bit aligned DRAM address - should start at a 256-bit aligned location)	19	0	Points to the first word of the first TFS line in DRAM.
0x44	TfsEndAdr (64-bit aligned DRAM address - should start at a 256-bit aligned location)	19	0	Points to the first word of the last TFS line in DRAM.
0x48	TfsFirstLineAdr (64-bit aligned DRAM address)	19	0	Points to the first word of the first TFS line to be encountered on the page. If the start of the page is in an inter-tag gap, then this value will be the same as *TFSStartAdr* since the first tag line reached will be the top line of a tag.
0x4C	DataRedun	1	0	Defines the data to redundancy ratio for the Reed Solomon encoder. Symbol size is always 4 bits, Codeword size is always 15 symbols (60 bits). 0 - 5 data symbols (20 bits), 10 redundancy symbols (40 bits) 1 -7 data symbols (28 bits), 8 redundancy symbols (32 bits)
0x50	Decode2DEn	1	0	Determines whether or not the data bits are to be 2D decoded rather than redundancy encoded (each 2 bits of the data bits becomes 4 output data bits). 0 = redundancy encode data 1 = decode each 2 bits of data into 4 bits
0x54	VariableDataPresent	1	0	Defines whether or not there is variable data in the tags. If there is none, no attempt is made to read tag data, and tag encoding should only reference fixed tag data.
0x58	EncodeFixed	1	0	Determines whether or not the lower 40 (or 56) bits of fixed data should be encoded into 120 bits or simply used as is.
0x5C	TagMaxDotpairs	8	0	The width of a tag in dot-pairs, minus 1. Minimum 0, Maximum=191.
0x60	TagMaxLine	9	0	The number of lines in a tag, minus 1. Minimum 0, Maximum = 383.
0x64	TagGapDot	14	0	The number of dot pairs between tags in the dot dimension minus 1. Only valid if *TagGapPresent*[bit 0] = 1.
0x68	TagGapLine	14	0	Defines the number of dotlines between tags in the line dimension minus 1. Only valid if *TagGapPresent*[bit1] = 1.
0x6C	DotPairsPerLine	14	0	Number of output dot pairs to generate per tag line.
0x70	DotStartTagSense	2	0	Determines for the first/even (bit 0) and second/odd (bit 1) rows of tags whether or not the first dot position of the line is in a tag. 1 = in a tag, 0 = in an inter-tag gap.

Table 126. TE Configuration Registers

Address TE_base+	register name	#bits	value on reset	description
0x74	TagGapPresent	2	0	Bit 0 is 1 if there is an inter-tag gap in the dot dimension, and 0 if tags are tightly packed. Bit 1 is 1 if there is an inter-tag gap in the line dimension, and 0 if tags are tightly packed.
0x78	YScale	8	1	Tag scale factor in Y direction. Output lines to the TFU will be generated YScale times.
0x80 to 0x84	DotStartPos	2x14	0	Determines for the first/even (0) and second/odd (1) rows of tags the number of dotpairs remaining minus 1, in either the tag or inter-tag gap at the start of the line.
0x88 to 0x8C	NumTags	2x8	0	Determines for the first/even and second/odd rows of tags how many tags are present in a line (equals number of tags minus 1).
Setup band related registers				
0xC0	NextBandStartTagDataAdr (64-bit aligned DRAM address - should start at a 256-bit aligned location)			Holds the value of StartTagDataAdr for the next band. This value is copied to StartTagDataAdr when DoneBand is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0xC4	NextBandEndOfTagData (64-bit aligned DRAM address)			Holds the value of EndOfTagData for the next band. This value is copied to EndOfTagData when DoneBand is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0xC8	NextBandFirstTagLineHeight	9	0	Holds the value of FirstTagLineHeight for the next band. This value is copied to FirstTagLineHeight when DoneBand gets is 1 and NextBandEnable is 1, or when Go transitions from 0 to 1.
0xCC	NextBandEnable			When NextBandEnable is 1 and DoneBand is 1, then when te_finishedband is set at the end of a band: -NextBandStartTagDataAdr is copied to StartTagDataAdr -NextBandEndOfTagData is copied to EndOfTagData -NextBandFirstTagLineHeight is copied to FirstTagLineHeight -DoneBand is cleared -NextBandEnable is cleared. *NextBandEnable* is cleared when *Go* is asserted.
Read-only band related registers				

Table 126. TE Configuration Registers

Address TE base+	register name	bits	value (at reset)	description
0xD0	DoneBand	1	0	Specifies whether the tag data interface has finished loading all the tag data for the band. It is cleared to 0 when *Go* transitions from 0 to 1. When the tag data interface has finished loading all the tag data for the band, the *te_finishedband* signal is given out and the *DoneBand* flag is set. If *NextBandEnable* is 1 at this time then *startTagDataAdr*, *endOfTagData* and *firstTaglineHeight* are updated with the values for the next band and *DoneBand* is cleared. Processing of the next band starts immediately. If *NextBandEnable* is 0 then the remainder of the TE will continue to run,, while the read control unit waits for *NextBandEnable* to be set before it restarts. Read only.
0xD4	StartTagDataAdr (64-bit aligned DRAM address - should start at a 256-bit aligned location)	19	0	The start address of the current row of raw tag data. This is initially points to the first word of the band's tag data, which should be aligned to a 128-bit boundary (i.e. the lower bit of this address should be 0). Read only.
0xD8	EndOfTagData (64-bit aligned DRAM address)	19	0	Points to the address of the final tag for the band. When all the tag data up to and including address *endOfTagData* has been read in, the *te_finishedband* signal is given and the doneBand flag is set. Read only.
0xDC	FirstTagLineHeight	9	0	The number of lines minus 1 in the first tag encountered in this band. This will be equal to *TagMaxLine* if the band starts at a tag boundary. Read only.
Work registers (set before starting the TE and must not be touched between bands)				
0x100	LineInTag	1	0	Determines whether or not the first line of the page is in a line of tags or in an inter-tag gap. 1 - in a tag, 0 - in an inter-tag gap.
0x104	LinePos	14	0	The number of lines remaining minus 1, in either the tag or the inter-tag gap in at the start of the page.
0x110 to 0x11C	TagData	4x32	0	This 128 bit register must be set up initially with the fixed data record for the page. This is either the lower 40 (or 56) bits (and the *encodeFixed* register should be set), or the lower 120 bits (and encodedFixed should be clear). The tagData[0] register contains the lower 32 bits and the tagData[3] register contains the upper 32 bits. This register is used throughout the tag encoding process to hold the next tag's variable data.
Work registers (set internally) **Read-only from the point of view of PCU register access**				
0x140	DotPos	14	0	Defines the number of dotpairs remaining in either the tag or inter-tag gap. Does not need to be setup.
0x144	CurrTagPlaneAdr	14	0	The dot-pair number being generated.
0x148	DotsInTag	1	0	Determines whether the current dot pair is in a tag or not 1 - in a tag, 0 - in an inter-tag gap.

Table 126. TE Configuration Registers

Address TE_base	register name	#bits	value on reset	description
0x14C	TagAltSense	1	0	Determines whether the production of output dots is for the first (and subsequent even) or second (and subsequent odd) row of tags.
0x154	CurrTFSAdr (64-bit aligned DRAM address)	19	0	Points to the start next line of the TFS to be read in.
0x158	ReadsRemaining	4	0	Number of reads remaining in the current burst from the raw tag data interface
0x15C	CountX	8	0	The number of tags remaining to be read (minus 1) by the raw tag data interface for the current line.
0x160	CountY	9	0	The number of times (minus 1) the tag data for the current line of tags needs to be read in by the raw tag data interface.
0x164	RtdTagSense	1	0	Determines whether the raw tag data interface is currently reading even rows of tags (=0) or odd rows of tags (=1) *with respect to the start of the page*. Note that this can be different from tagAltSense since the raw tag data interface is reading ahead of the production of dots.
0x168	RawTagDataAdr (64-bit aligned DRAM address)	19	0	The current read address within the unencoded raw tag data.

The PCU accessible registers are divided amongst the TE top level and the TE sub-blocks. This is achieved by including write decoders in the sub-blocks as well as the top level, see Figure 153. In order to perform reads the sub-block registers are fed to the top level where the read decode is carried out on all the PCU accessible TE registers.
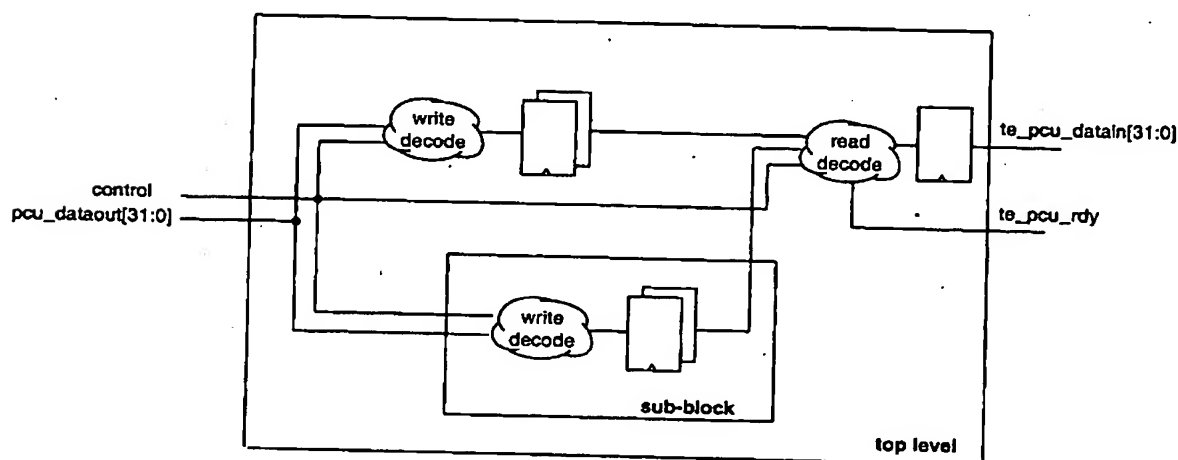
Figure 153. Block diagram of PCU accesses

26.6.5.1 Starting the TE and restarting the TE between bands

The TE must be started after the TFU.

For the first band of data, users set up *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBand-FirstTagLineHeight* as well as other TE configuration registers. Users then set the TE's *Go* bit to start processing of the band. When the tag data for the band has finished being decoded, the *te_finishedband* interrupt will be sent to the PCU and ICU indicating that the memory associated with the first band is now free. Processing can now start on the next band of tag data.

In order to process the next band *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* need to be updated before writing a 1 to *NextBandEnable*. There are 4 mechanisms for restarting the TE between bands:

    a. *te_finishedband* causes an interrupt to the CPU. The TE will have set its *DoneBand* bit. The CPU reprograms the *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers, and sets *NextBandEnable* to restart the TE.

    b. The CPU programs the TE's *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBand-FirstTagLineHeight* registers and sets the *NextBandEnable* flag before the end of the current band. At the end of the current band the TE sets *DoneBand*. As *NextBandEnable* is already 1, the TE starts processing the next band immediately.

    c. The PCU is programmed so that *te_finishedband* triggers the PCU to execute commands from DRAM to reprogram the *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers and set the *NextBandEnable* bit to start the TE processing the next band. The advantage of this scheme is that the CPU could process band headers in advance and store the band commands in DRAM ready for execution.

    d. This is a combination of *b* and *c* above. The PCU (rather than the CPU in *b*) programs the TE's *NextBandStartTagDataAdr*, *NextBandEndTagData* and *NextBandFirstTagLineHeight* registers and sets the *NextBandEnable* bit before the end of the current band. At the end of the current band the TE sets *DoneBand* and pulses *te_finishedband*. As *NextBandEnable* is already 1, the TE starts processing the next band immediately. Simultaneously, *te_finishedband* triggers the PCU to fetch commands from DRAM. The TE will have restarted by the time the PCU has fetched commands from DRAM. The PCU commands program the TE next band shadow registers and sets the *NextBandEnable* bit.

After the first tag on the page, all bands have their first tag start at the top i.e. *NextBandFirstTagLineHeight* = *TagMaxLine*. Therefore the same value of *NextBandFirstTagLineHeight* will normally be used for all bands. Certainly, *NextBandFirstTagLineHeight* should not need to change after the second time it is programmed.

26.6.6 TE Top Level FSM

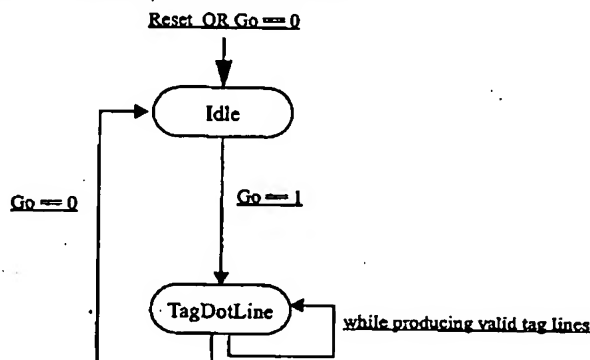The following diagram illustrates the states in the FSM.

Figure 154. Tag Encoder Top-Level FSM

At the highest level, the TE state machine steps through the output lines of a page one line at a time, with the starting position either in an inter-tag gap (signal *dotsintag* = 0) or in a tag (signals *tfsvalid* and *tdvalid* and *lineintag* = 1) (a SoPEC may be only printing part of a tag due to multiple SoPECs printing a single line).

If the current position is within an inter-tag gap, an output of 0 is generated. If the current position is within a tag, the tag format structure is used to determine the value of the output dot, using the appropriate encoded data bit from the fixed or variable data buffers as necessary. The TE then advances along the line of dots, moving through tags and inter-tag gaps according to the tag placement parameters.

Table 127 highlights the signals used within the FSM.

Table 127. Signals used within TE top level FSM

Signal Name	Function
pclk	Sync clock used to register all data within the FSM
prst_n, te_reset	Reset signals
advtagline	1 cycles pulse indicating to TDI and TFS sub-blocks to move onto the next line of Tag data
currdotlineadr[13:0]	Address counter starting 2 pclk ahead of currtagplaneadr to generate the correct dotpair for the current line
dotpos	Counter to identify how many dotpairs wide the tag/gap is
dotsintag	Signal identifying whether the dotpair are in a tag(1)/gap(0)
lineintag_temp	Identical to lineintag but generated 1 pclk earlier
linepos_shadow	Shadow register for linepos due to linepos being written to by 2 different processes
tataltsense	Flag which alternates between tag/gap lines
te_state	FSM state variable
teplanebuf	6-bit shift register used to format dotpairs into a byte for the TFU
wradvline	Advance line signal strobed when the last byte in a line is placed on te_tfu_wdata

Due to the 2 *system clock* delay in the TFS (both Table A and Table B outputs are registered) the TE FSM is working 2 *system clock* cycles **AHEAD** of the logic generating the write data for the TFU. As a result the following control signals had to be single/double registered on the *system clock*.
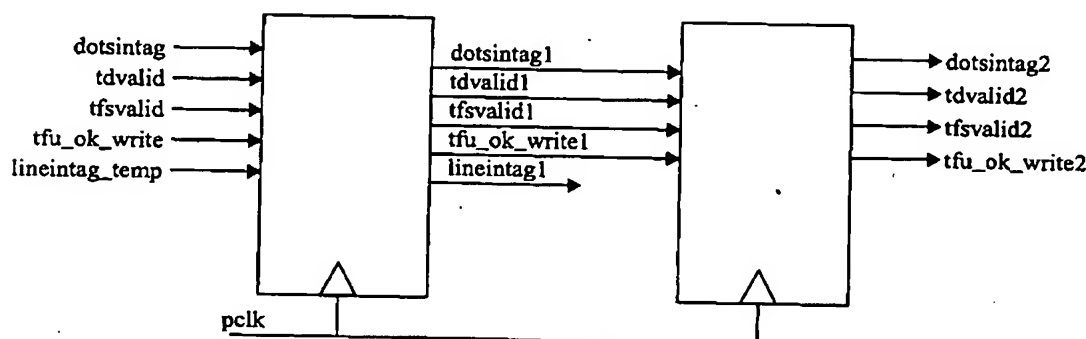

Figure 155. Generated Control Signals

The *tag_dot_line* state can be broken down into 3 different stages.

Stage1:- The state *tag_dot_line* is entered due to the *go* signal becoming active. This state controls the writing of dotbytes to the TFU. As long as the tag line buffer address is not equal to the *dotpairsperline* register value and *tfu_te_oktowrite* is active, and there is valid TFS and TD available or taggaps, dotpairs are buffered into bytes and written to the TFU. The tag line buffer address is used internally but not supplied to the TFU since the TFU is a FIFO rather than the line store used in PEC1.

While generating the dotline of a tag/gap line (*lineintag* flag = 1) the dot position counter *dotpos* is decremented/reloaded (with tagmaxdotpairs or taggapdot) as the TE moves between tags/gaps. The *dotsintag* flag is toggled between tags/gaps (0 for a gap, 1 for a tag). This pattern continues until the end of a dotline approaches (*currdotlineadr* == *dotpairsperline*).

2 *system clock* cycles before the end of the dotline the *lineintag* and *tagaltsense* signals must be prepared for the next dotline be it in a tag/gap dotline or a purely gap dotline.

Stage2:- At this point the end of a dot line is reached so it is time to decrement the *linepos* counter if still in a tag/gap row or reload the *linepos* register, *dotpos* counter and reprogram the *dotsintag* flag if going onto another tag/gap or pure gap row. Any signal with the _temp extension means this register is updated a cycle early in order for the real register to get its correct value while switching between dot lines and tag rows when *dotpos* and *linepos* counters reach zero i.e when *dotpos* = 0 the end of a tag/gap has been reached, when *linepos* = 0 the end of a tag row is reached. This stage uses the signals *lineintag_temp* and *tagaltsense* which were generated one *system clock* cycle earlier in Stage 1.

Stage3:- This stage implements the writing of dotpairs to the correct part of the 6-bit shift register based on the LSBs of *currtagplaneadr* and also implements the counter for the *currtagplaneadr*. The *currtagpla-neadr* is reset on reaching *currtagplaneadr* = (*dotpairsperline* - 1). All the qualifier signals e.g *dotsintag* for this stage are delayed by 2 *system clock* cycles i.e. the *currtagplaneadr* (which is the internal write address not needed by the TFU) cannot be incremented until the dotpairs are available which is always 2 *system clock* cycles later than when *currdotlineadr* is incremented.

The *wradvline* and *advtagline* pulses are generated using the same logic (currently separated in the PEC1 Tag Encoder VHDL for clarity). Both of these pulses used to update further registers hence the reason they do not use the delayed by 2 *system clock* cycle qualifiers.

26.6.7 Combinational Logic

The TDI is responsible for providing the information data for a tag while the TFSI is responsible for deciding whether a particular dot on the tag should be printed as background pattern or tag information. Every dot within a tag's boundary is either an information dot or part of the background pattern.
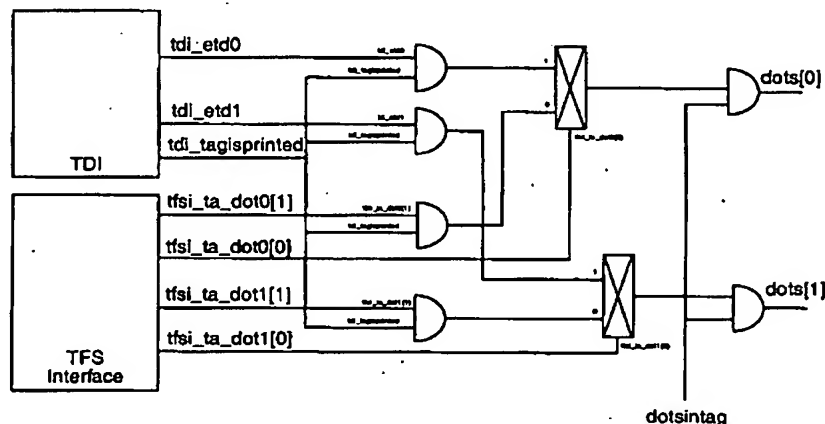
Figure 156. Logic to combine dot information and Encoded Data

The resulting lines of dots are stored in the TFU.

The TFSI reads one Tag Line Structure (TLS) from the DIU for every dot line of tags. Depending on the current printing position within the tag (indicated by the signal *tagdotnum*), the TFS interface outputs dot information for two dots and if necessary the corresponding read addresses for encoded tag data. The read address are supplied to the TDI which outputs the corresponding data values.

These data values (*tdi_etd0* and *tdi_etd1*) are then combined with the dot information (*tfsi_ta_dot0* and *tfsi_ta_dot1*) to produce the dot values that will actually be printed on the page (*dots*), see Figure 156.
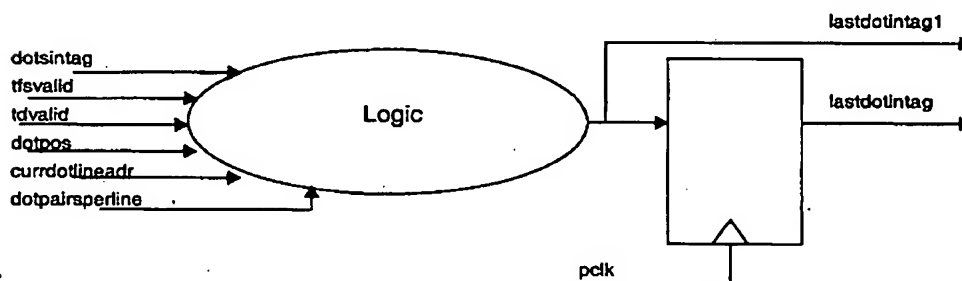
Figure 157. Generation of Lastdotintag/1

The signal *lastdotintag* is generated by checking that the dots are in a tag (*dotsintag* = 1) and that the dot-position counter dotpos is equal to zero. It is also used by the TFS to load the index address register with zeros at the end of a tag as this is always the starting index when going from one tag to the next. *lastdotin-*

tag is gated with *advtagline* in the TFSi (Table C) where *adv_tfs_line* pulse is used to update the Table C address reg for the new tag line - this is because *lastdotintag* occurs a cycle earlier than *adv_tfs_line* which would result in the wrong Table C value for the last dotpair. *lastdotintag* is also used in the TDi FSM (etd_switch state) to pulse the *etd_advtag* signal hence switching buffers in the ETDi for the next tag.

The signal *lastdotintag1* is identical to *lastdotintag* except it is combinatorially generated (1 cycle earlier than *lastdotintag*, except at the end of a *tagline*). *lastdotintag1* signal is only used in the TDi to reset the *tdvalid* signal on the cycle when *dotpos* = 0. Note the UNSIGNED(*currdotlineadr*) = UNSIGNED(*dotpairsperline*) - 1 not UNSIGNED(*currdotlineadr*) = UNSIGNED(*dotpairsperline*) - 2 as in the *lastdotintag_gen* process as this is an combinatorial process.
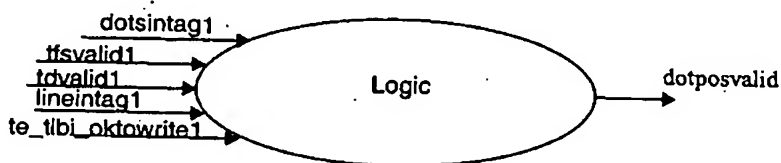
Figure 158. Generation of Dot Position Valid

The *dotposvalid* signal is created based on being in a tag line (*lineintag1* = 1), dots being in a tag (*dotsintag1* = 1), having a valid tag format structure available (*tfsvalid1* = 1) and having encoded tag data available (*tdvalid1* = 1). Note that each of the qualifier signals are delayed by 1 *pclk* cycle due to the registering of Table A output data into Table C where *dotposvalid* is used. The *dotposvalid* signal is used as an enable to load the Table C address register with the next index into Table B which in turn provides the 2 addresses to make 2 dots available.
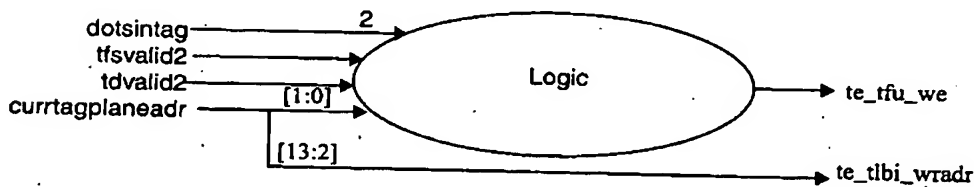
Figure 159. Generation of write enable to the TFU

The signal *te_tfu_wdatavalid* can only be active if in a taggap or if valid tag data is available (*tdvalid2* and *tfsvalid2*) and the *currtagpplaneadr*(1:0) equal 11 i.e. a byte of data has been generated by combining four dotpairs.
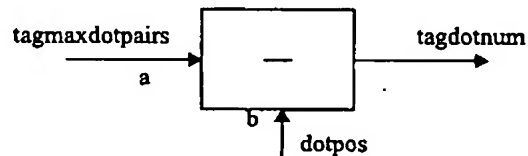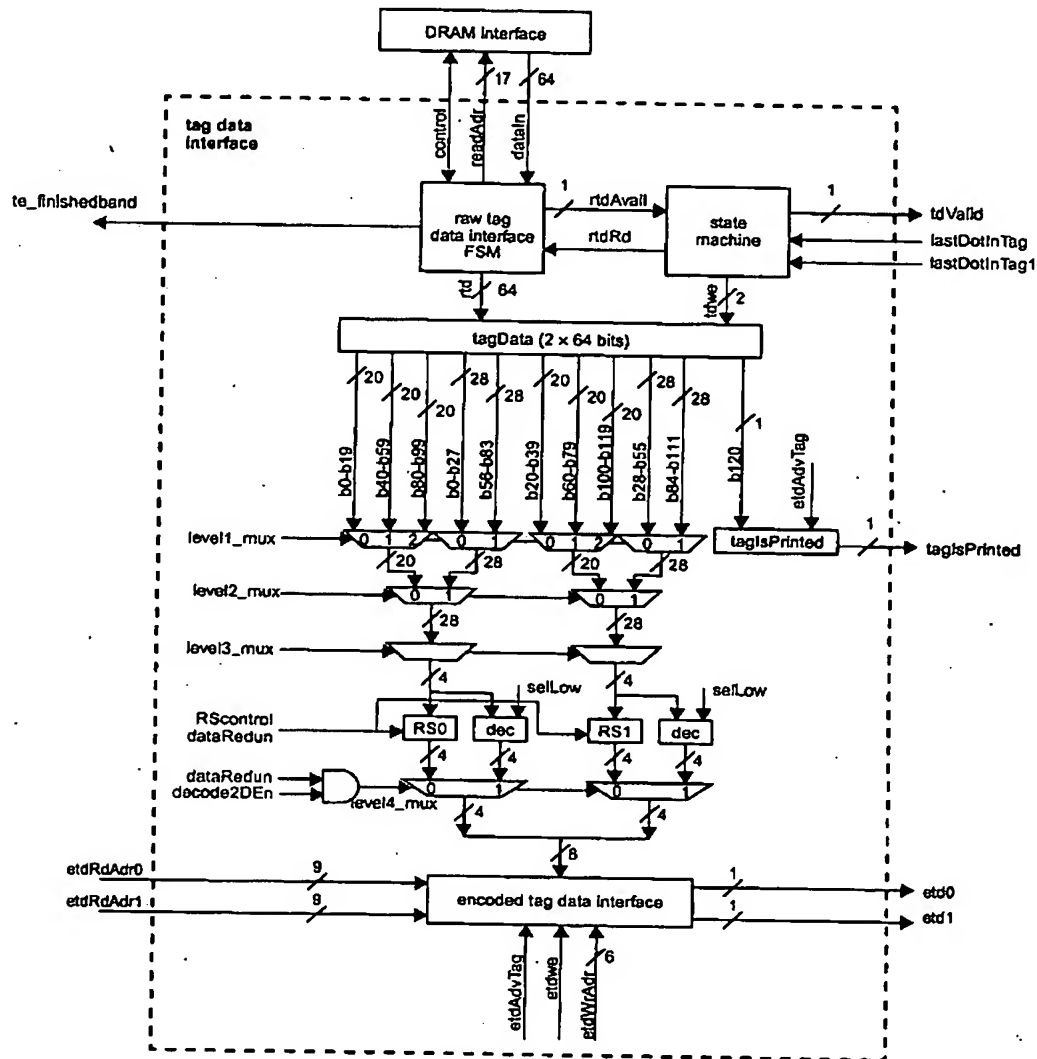

Figure 160. Generation of Tag Dot Number

The signal *tagdotnum* tells the TFS how many dotpairs remain in a tag/gap. It is calculated by subtracting the value in the *dotpos* counter from the value programmed in the *tagmaxdotpairs* register.

26.7   TAG DATA INTERFACE (TDI)

26.7.1   I/O Specification

Table 128. TDI Port List

Signal name	I/O	Description
Clocks and Resets		
pclk	In	SoPEC system clock
prst_n	In	Active-low, synchronous reset in pclk domain.
DIU Read Interface Signals		
diu_data[63:0]	In	Data from DRAM.
td_diu_rreq	Out	Data request to DRAM.
td_diu_radr[21:5]	Out	Read address to DRAM.
diu_td_rack	In	Data acknowledge from DRAM.
diu_td_rvalid	In	Data valid signal from DRAM.
PCU Interface Data, Control Signals and		
pcu_dataout[31:0]	In	PCU writes this data.
pcu_addr[8:2]	In	PCU accesses this address.
pcu_rwn	In	Global read/write-not signal from PCU.
pcu_te_sel	In	PCU selects TE for r/w access.
pcu_te_reset	In	PCU reset.
td_te_doneband td_te_dataredun td_te_decode2den td_te_variabledatapresent td_te_encodefixed td_te_numtags0 td_te_numtags1 td_te_starttagdataadr td_te_rawtagdataadr td_te_endoftagdata td_te_firsttaglineheight td_te_tagdata0 td_te_tagdata1 td_te_tagdata2 td_te_tagdata3 td_te_countx td_te_county td_te_rtdtagsense td_te_readsremaining	Out	PCU readable registers.
TFS (Tag Format Structure)		
tfsi_adr0[8:0]	In	Read address for dot0
tfsi_adr1[8:0]	In	Read address for dot1
Bandstore Signals		
cdu_startofbandstore[24:0]	In	Start memory area allocated for page bands
cdu_endofbandstore[24:0]	In	Last address of the memory allocated for page bands
te_finishedband	Out	Tag encoder band finished

Figure 161. TDI Architecture

26.7.2 Introduction

The tag data interface is responsible for obtaining the raw tag data and encoding it as required by the tag encoder. The smallest typical tag placement is 2mm × 2mm, which means a tag is at least 126 1600 dpi dots wide.

In PEC1, in order to keep up with the HCU which processes 2 dots per cycle, the tag data interface has been designed to be capable of encoding a tag in 63 cycles. This is actually accomplished in approximately 52 cycles within PEC1. For SoPEC the TE need only produce one dot per cycle; it should be able to produce tags in no more than twice the time taken by the PEC1 TE. Moreover, any change in implementation

from two dots to one dot per cycle should not lose the 63/52 cycle performance edge attained in the PEC1 TE.

As shown in Figure 162, the tag data interface contains a raw tag data interface FSM that fetches tag data from DRAM, two symbol-at-a-time GF(2^4) Reed-Solomon encoders, an encoded data interface and a state machine for controlling the encoding process. It also contains a *tagData* register that needs to be set up to hold the fixed tag data for the page.

The type of encoding used depends on the registers *TE_encodefixed*, *TE_dataredun* and *TE_decode2den* the options being,

- (15,5) RS coding, where every 5 input symbols are used to produce 15 output symbols, so the output is 3 times the size of the input. This can be performed on **fixed and variable tag data**.
- (15,7) RS coding, where every 7 input symbols are used to produce 15 output symbols, so for the same number of input symbols, the output is not as large as the (15,5) code (for more details see section 26.7.6 on page 400). This can be performed on **fixed and variable tag data**.
- 2D decoding, where each 2 input bits are used to produce 4 output bits. This can be performed on **fixed and variable tag data**.
- no coding, where the data is simply passed into the Encoded Data Interface. This can be performed on **fixed data only**.

Each tag is made up of **fixed tag data** (i.e. this data is the same for each tag on the page) and **variable tag data** (i.e. different for each tag on the page).

Fixed tag data is either stored in DRAM as 120-bits when it is already coded (or no coding is required), 40-bits when (15,5) coding is required or 56-bits when (15,7) coding is required. Once the **fixed tag data** is coded it is 120-bits long. It is then stored in the Encoded Tag Data Interface.

The **variable tag data** is stored in the DRAM in uncoded form. When (15,5) coding is required, the 120-bits stored in DRAM are encoded into 360-bits. When (15,7) coding is required, the 112-bits stored in DRAM are encoded into 240-bits. When 2D decoding is required the 120-bits stored in DRAM are converted into 240-bits. In each case the encoded bits are stored in the Encoded Tag Data Interface.

The encoded **fixed and variable tag data** are eventually used to print the tag.

The **fixed tag data** is loaded in once from the DRAM at the start of a page. It is encoded as necessary and is then stored in one of the 8x15-bits registers/RAMs in the Encoded Tag Data Interface. This data remains unchanged in the registers/RAMs until the next page is ready to be processed.

The 120-bits of unencoded **variable tag data** for each tag is stored in four 32-bit words. The TE re-reads the **variable tag data**, for a particular tag from DRAM, every time it produces that tag. The **variable tag data** FIFO which reads from DRAM has enough space to store 4 tags.

26.7.3   Data Flow

An overview of the dataflow through the TDI can be seen in Figure 162 below.
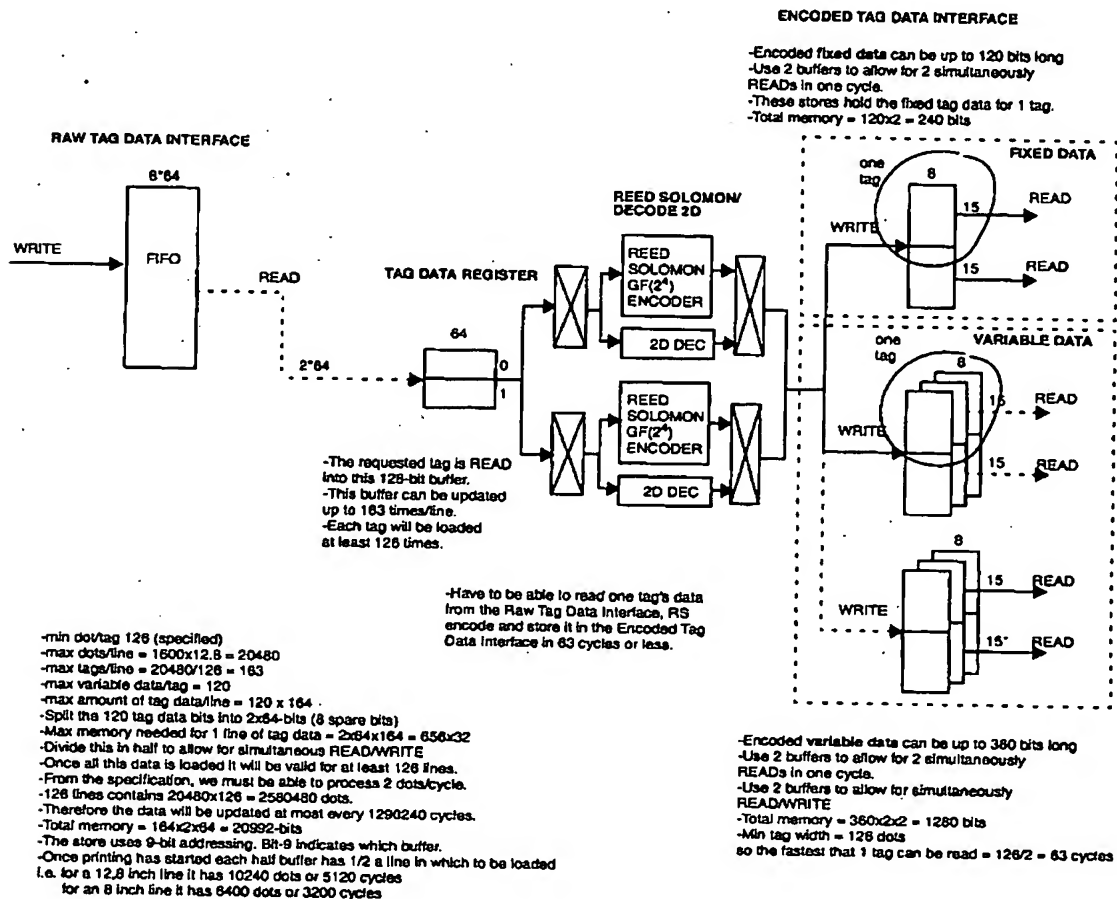

ENCODED TAG DATA INTERFACE
-Encoded fixed data can be up to 120 bits long
-Use 2 buffers to allow for 2 simultaneously
READs in one cycle.
-These stores hold the fixed tag data for 1 tag.
-Total memory = 120x2 = 240 bits

RAW TAG DATA INTERFACE

8*64

FIFO

WRITE

READ

TAG DATA REGISTER

REED SOLOMON/ DECODE 2D

REED SOLOMON GF(2⁴) ENCODER

2D DEC

REED SOLOMON GF(2⁴) ENCODER

2D DEC

64

2*64

0
1

FIXED DATA

one tag

8

15   READ

WRITE

15   READ

VARIABLE DATA

one tag

8

15   READ

WRITE

15   READ

8

15   READ

WRITE

15   READ

-The requested tag is READ into this 128-bit buffer.
-This buffer can be updated up to 163 times/line.
-Each tag will be loaded at least 126 times.

-Have to be able to read one tag's data from the Raw Tag Data Interface, RS encode and store it in the Encoded Tag Data Interface in 63 cycles or less.

-min dot/tag 126 (specified)
-max dots/line = 1600x12.8 = 20480
-max tags/line = 20480/126 = 163
-max variable data/tag = 120
-max amount of tag data/line = 120 x 164
-Split the 120 tag data bits into 2x64-bits (8 spare bits)
-Max memory needed for 1 line of tag data = 2x64x164 = 656x32
-Divide this in half to allow for simultaneous READ/WRITE
-Once all this data is loaded it will be valid for at least 126 lines.
-From the specification, we must be able to process 2 dots/cycle.
-126 lines contains 20480x126 = 2580480 dots.
-Therefore the data will be updated at most every 1290240 cycles.
-Total memory = 164x2x64 = 20992-bits
-The store uses 9-bit addressing. Bit-9 indicates which buffer.
-Once printing has started each half buffer has 1/2 a line in which to be loaded
i.e. for a 12.8 inch line it has 10240 dots or 5120 cycles
    for an 8 inch line it has 6400 dots or 3200 cycles

-Encoded variable data can be up to 380 bits long
-Use 2 buffers to allow for 2 simultaneously
READs in one cycle.
-Use 2 buffers to allow for simultaneous
READ/WRITE
-Total memory = 380x2x2 = 1280 bits
-Min tag width = 126 dots
so the fastest that 1 tag can be read = 126/2 = 63 cycles

Figure 162.   Data Flow Through the TDI

The TD interface consists of the following main sections:
- the Raw Tag Data Interface - fetches tag data from DRAM;
- the tag data register;
- 2 Reed Solomon encoders - each encodes one 4-bit symbol at a time;
- the Encoded Tag Data Interface - supplies encoded tag data for output;
- Two 2D decoders.

The main performance specification for PEC1 is that the TE must be able to output data at a continuous rate of 2 dots per cycle.

26.7.4 Raw tag data Interface

The raw tag data interface (RTDI) provides a simple means of accessing raw tag data in DRAM. The RTDI passes tag data into a FIFO where it can be subsequently read as required. The 64-bit output from the FIFO can be read directly, with the value of the *wr_rd_counter* being used to set/reset as the enable signal (*rtdAvail*). The FIFO is clocked out with receipt of an *rtdRd* signal from the TS FSM.

Figure 163 shows a block diagram of the raw tag data interface.
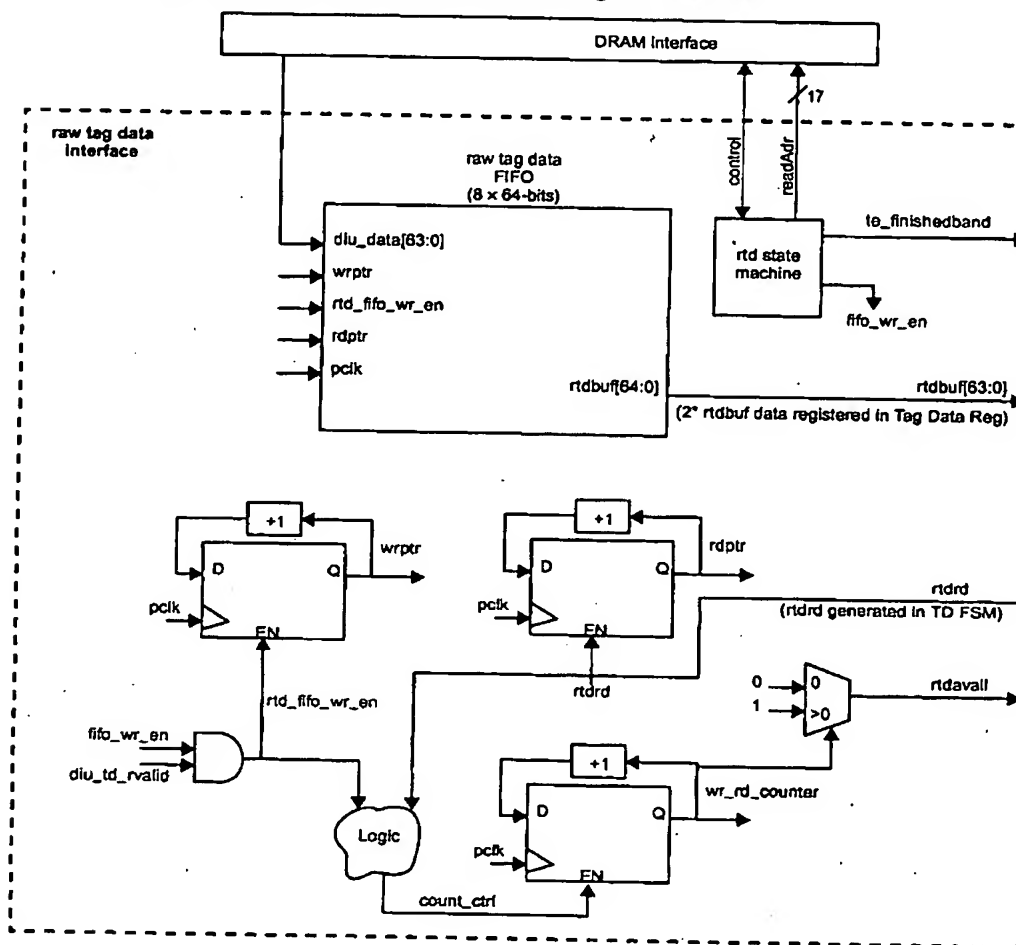
Figure 163. Raw tag data Interface block diagram

26.7.4.1 RTDI FSM

The RTDI state machine is responsible for keeping the raw tag FIFO full. The state machine reads the line of tag data once for each printline that uses the tag. This means a given line of tag data will be read at least 126 times since the tag height is 126 lines for 2 mm tags. Note that the first line of tag data may be read fewer than 126 times since the start of the page may be within a tag. In addition odd and even rows of tags may contain different numbers of tags.

Section 26.6.5.1 outlines how to start the TE and restart it between bands. Users must set the *NextBand-StartTagDataAdr, NextBandEndOfTagData, NextBandFirstTagLineHeight and numTags[0], numTags[* registers before starting the TE by asserting Go.

To restart the tag encoder for second and subsequent bands of a page, the *NextBandStartTagDataAdr, NextBandEndOfTagData* and *NextBandFirstTagLineHeigh* registers need to be updated (typically *numTags[0]* and *numTags[1]* will be the same if the previous band contains an even number of tag rows) and *NextBandEnable* set. See Section 26.6.5.1 for a full description of the four ways of reprogramming the TE between bands.

The tag data is read once for every printline containing tags. When maximally packed, a row of tags contains 163 tags (see Table 121 on page 364).

The RTDI State Flow diagram is shown in Figure 164. An explanation of the states follows:

idle state:- Stay in the idle state if there is no variable data present. If there is variable data present and there are at least 4 spaces left in the FIFO then request a burst of 2 tags from the DRAM (1 * 256bits). Counter *countx* is assigned the number of tags in a even/odd line which depends on the value of register *rtdtagsense*. Down-counter *county* is assigned the number of dot lines high a tag will be (min 126). Initially it must be set the *firsttaglineheight* value as the TE may be between pages (i.e. a partial tag). For normal tag generation *county* will take the value of *tagmaxline* register.

diu_access:- The diu_access state will generate a request to the DRAM if there are at least 4 spaces in the FIFO. This is indicated by the counter *wr_rd_counter* which is incremented/decremented on writes/reads of the FIFO. As long as *wr_rd_counter* is less than 4 (FIFO is 8 high) there must be 4 locations free. A control signal called *td_diu_radrvalid* is generated for the duration of the DRAM burst access. Addresses are sent in bursts of 1. The counter *burst_count* controls this signal, (will involve modification to existing TE Code.)

If there is an odd number of tags in line then the last DRAM read will contain a tag in the first 128 bits and padding in the final 128 bits.

fifo_load:- This state controls the addressing to the DRAM. Counters *countx* and *county* are used to monitor whether the TE is processing a line of dots within a row of tags. When *countx* is zero it means all tag dots for this row are complete. When *county* is zero it means the TE is on the last line of dots (prior to Y scaling) for this row of tags. When a row of tags is complete the sense of *rtdtagsense* is inverted (odd/even). The *rawtagdataadr* is compared to the *te_endoftagdata* address. If *rawtagdataadr = endoftagdata* the *doneband* signal is set, the *finishedband* signal is pulsed, and the FSM enters the *rtd_stall* state until the *doneband* signal is reset to zero by the PCU by which time the *rawtagdata, endoftagedata* and *firsttaglineheight* registers are setup with new values to restart the TE. This state is used to count the 64-bit reads from the DIU. Each time *diu_td_rvalid* is high *rtd_data_count* is incremented by 1. The compare of *rtd_data_count = rtd_num* is neccessary to find out when either all 4*64-bit data has been received or n*64-bit data (depending on a match of *rawtagdataadr = endoftagdata* in the middle of a set of 4*64-bit values being returned by the DIU.

rtd_stall:- This state waits for the the *doneband* signal to be reset (see page 379 for a description of how this occurs). Once reset the FSM returns to the idle state. This states also performs the same count on the

diu_data read as above in the case where *diu_td_rvalid* has not gone high by the time the addressing is complete and the end of band data has been reached i.e. *rawtagdataadr = endoftagdata*
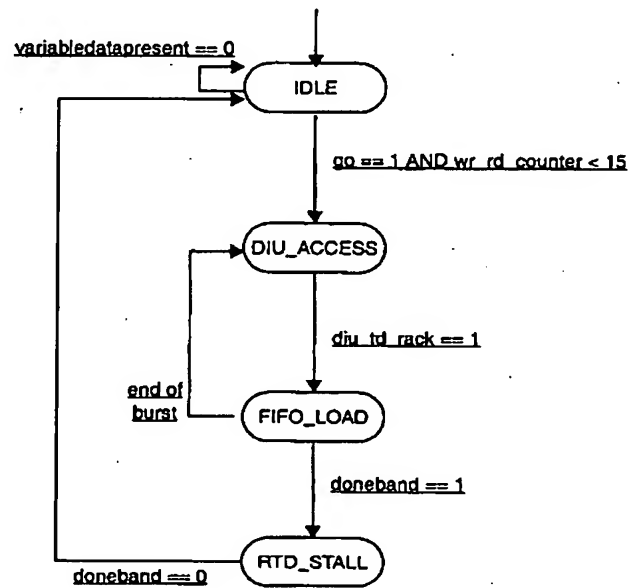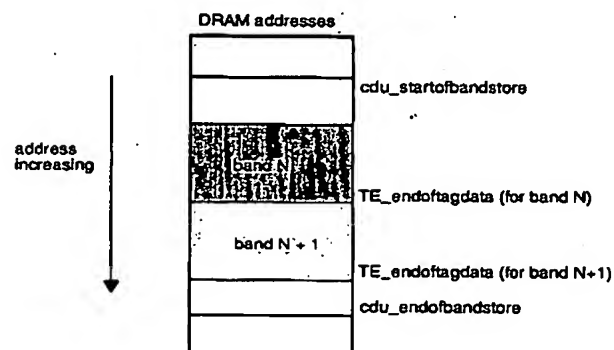

Figure 164. RTDI State Flow Diagram

Figure 165. Relationship between TE_endoftagdata, cdu_startofbandstore and cdu_endofbandstore

26.7.5   TDI state machine

The tag data state machine has two processing phases. The first processing phase is to encode the fixed tag data stored in the 128-bit (2 × 64-bit) tag data register. The second is to encode tag data as it is required by the tag encoder.

When the Tag Encoder is started up, the fixed tag data is already preloaded in the 128 bit tag data record. If *encodeFixed* is set, then the 2 codewords stored in the lower bits of the tag data record need to be encoded: 40 bits if *dataRedun* = 0, and 56 bits if *dataRedun* = 1. If *encodeFixed* is clear, then the lower 120 bits of the tag data record must be passed to the encoded tag data interface without being encoded.

When *encodeFixed* is set, the symbols derived from codeword 0 are written to codeword 6 and the symbols derived from codeword 1 are written to codeword 7. The data symbols are stored first and then the remaining redundancy symbols are stored afterwards, for a total of 15 symbols. Thus, when *dataRedun* = 0, the 5 symbols derived from bits 0-19 are written to symbols 0-4, and the redundancy symbols are written to symbols 5-14. When *dataRedun* = 1, the 7 symbols derived from bits 0-27 are written to symbols 0-6, and the redundancy symbols are written to symbols 7-14.

When *encodeFixed* is clear, the 120 bits of fixed data is copied directly to codewords 6 and 7.

The TDI State Flow diagram is shown in Figure 166. An explanation of the states follows.

Figure 166. TDI State Flow Diagram

idle:- In the idle state wait for the tag encoder *go* signal - *top_go* = 1. The first task is to either store or encode the Fixed data. Once the Fixed data is stored or encoded/stored the *donefixed* flag is set. If there is

no variable data the FSM returns to the idle state hence the reason to check the *donefixed* flag before advancing i.e. only store/encode the fixed data once.

fixed_data:- In the fixed_data state the FSM must decode whether to directly store the fixed data in the ETDi or if the fixed data needs to be either (15:5) (40-bits) or (15:7) (56-bits) RS encoded or 2D decoded. The values stored in registers *encodefixed* and *dataredun* and *decode2den* determine what the next state should be.

bypass_to_etdi:- The bypass_to_etdi takes 120-bits of fixed data(pre-encoded) from the *tag_data(127:0)* register and stores it in the 15*8 (by 2 for simultaneous reads) buffers. The data is passed from the *tag_data* register through 3 levels of muxing (level1, level2, level3) where it enters the RS0/RS1 encoders (which are now in a straight through mode (i.e. *control_5* and *control_7* are zero hence the data passes straight from the input to the output). The MSBs of the *etd_wr_adr* must be high to store this data as code-words 6,7.

etd_buf_switch:- This state is used to set the *tdvalid* signal and pulse the *etd_adv_tag* signal which in turn is used to switch the read write sense of the ETDi buffers (*wrsb0*). The *firsttime* signal is used to identify the first time a tag is encoded. If zero it means read the tag data from the RTDi FIFO and encode. Once encoded and stored the FSM returns to this state where it evaluates the sense of *tdvalid*. First time around it will be zero so this sets *tdvalid* and returns to the readtagdata state to fill the 2nd ETDi buffer. After this the FSM returns to this state and waits for the *lastdotintag* signal to arrive. In between tags when the last-dotingtag signal is received the *etd_adv_tag* is pulsed and the FSM goes to the readtagdata state. However if the *lastdotintag* signal arrives at the end of a line there is an extra 1 cycle delay introduced in generating the *etd_adv_tag* pulse (via *etd_adv_tag_endofline*) due to the pipelining in the TFS. This allows all the previous tag to be read from the correct buffer and seamless transfer to the other buffer for the next line.

readtagdata:- The readtagdata state waits to receive a *rtdavail* signal from the raw tag data interface which indicates there is raw tag data available. The *tag_data* register is 128-bits so it takes 2 pulses of the *rtdrd* signal to get the 2*64-bits into the *tag_data* register. If the *rtdavail* signal is set *rtdrd* is pulsed for 1 cycle and the FSM steps onto the loadtagdata state. Initially the flag *first64bits* will be zero. The 64-bits of *rtd* are assigned to the *tag_data[63:0]* and the flag *first64bits* is set to indicate the first raw tag data read is complete. The FSM then steps back to the read_tagdata state where it generates the second *rtdrd* pulse. The FSM then steps onto the loadtagdata state for where the second 64-bits of rawtag data are assigned to *tag_data[128:64]*.

loadtagdata:- The loadtagdata state writes the raw tag data into the *tag_data* register from the RTDi FIFO. The *first64bits* flag is reset to zero as the *tag_data* register now contains 120/112 bits of variable data. A decode of whether to (15:5) or (15:7) RS encode or 2D decode this data decides the next state.

rs_15_5:- The rs_15_5 (Reed Solomon (15:5) mode) state either encodes 40-bit Fixed data or 120-bit Variable data and provides the encoded tag data write address and write enable (*etd_wr_adr* and *etdwe* respectively). Once the fixed tag data is encoded the *donefixed* flag is set as this only needs to be done once per page. The *variabledatapresent* register is then polled to see if there is variable data in the tags. If there is variable data present then this data must be read from the RTDi and loaded into the *tag_data* register. Else the *tdvalid* flag must be set and FSM returns to the idle state. control_5 is a control bit for the RS Encoder and controls feedforward and feedback muxes that enable (15:5) encoding.

The rs_15_5 state also generates the control signals for passing 120-bits of variable tag data to the RS encoder in 4-bit symbols per clock cycle. *rs_counter* is used both to control the level1_mux and act as the 15-cycle counter of the RS Encoder. This logic cycles for a total of 3*15 cycles to encode the 120-bits.

rs_15_7:- The rs_15_7 state is similar to the rs_15_5 state except the level1_mux has to select 7 4-bit symbols instead of 5.

decode_2d_15_5, decode_2d_15_7:- The decode_2d states provides the control signals for passing the 120-bit variable data to the 2D decoder. The 2 lsbs are decoded to create 4 bits. The 4 bits from each

decoder are combined and stored in the ETDi. Next the 2 MSBs are decoded to create 4 bits. Again the 4 bits from each decoder are combined and stored in the ETDi.

As can be seen from Figure 161 on page 386 there are 3 stages of muxing between the Tag Data register and the RS encoders or 2D decoders. Levels 1-2 are controlled by *level1_mux* and *level2_mux* which are generated within the TDi FSM as is the write address to the ETDi buffers (*etd_wr_adr*)

Figures 167 through 172 illustrate the mappings used to store the encoded fixed and variable tag data in the ETDI buffers.
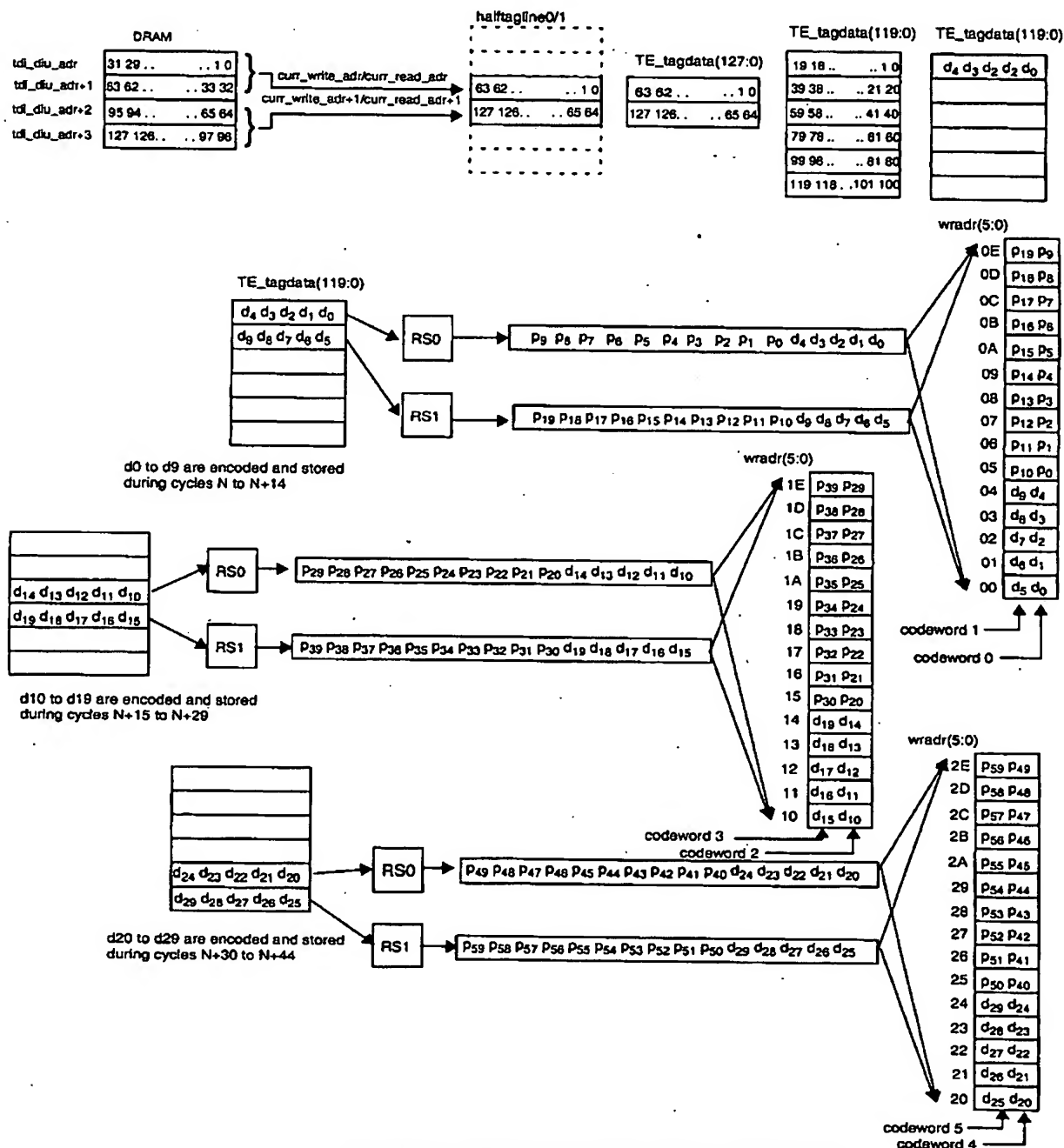
Figure 167. Mapping of the tag data to codewords 0-7
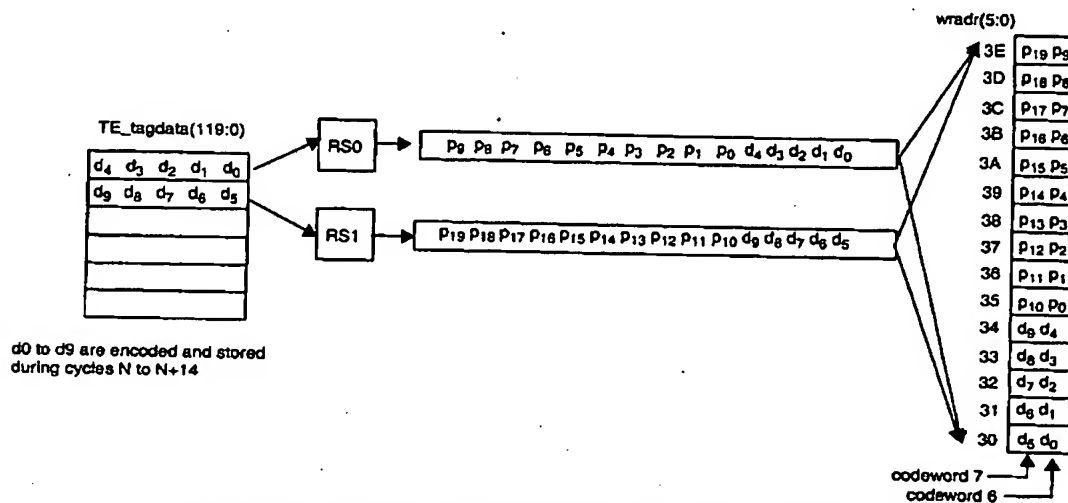
Figure 168. Coding and mapping of uncoded Fixed Tag Data for (15,5) RS encoder
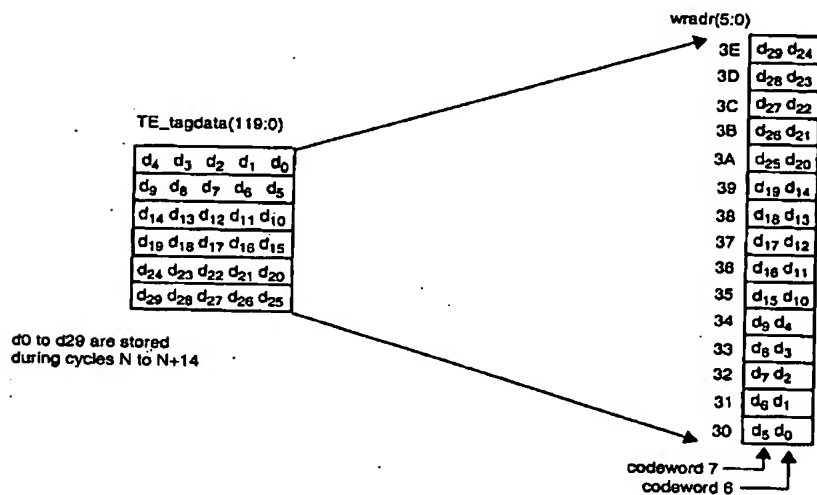
Figure 169. Mapping of pre-coded Fixed Tag Data
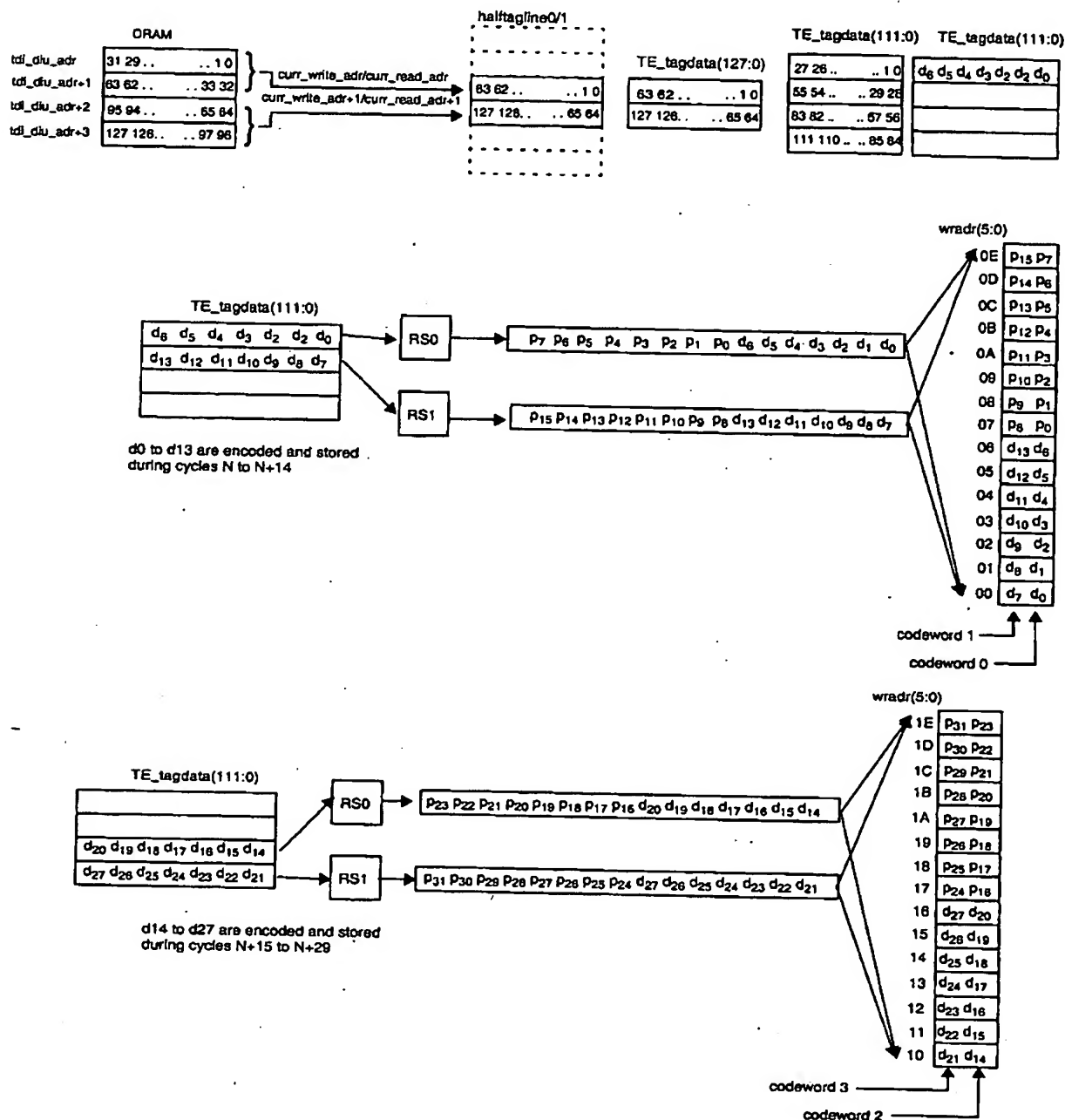
Figure 170. Coding and mapping of Variable Tag Data for (15,7) RS encoder
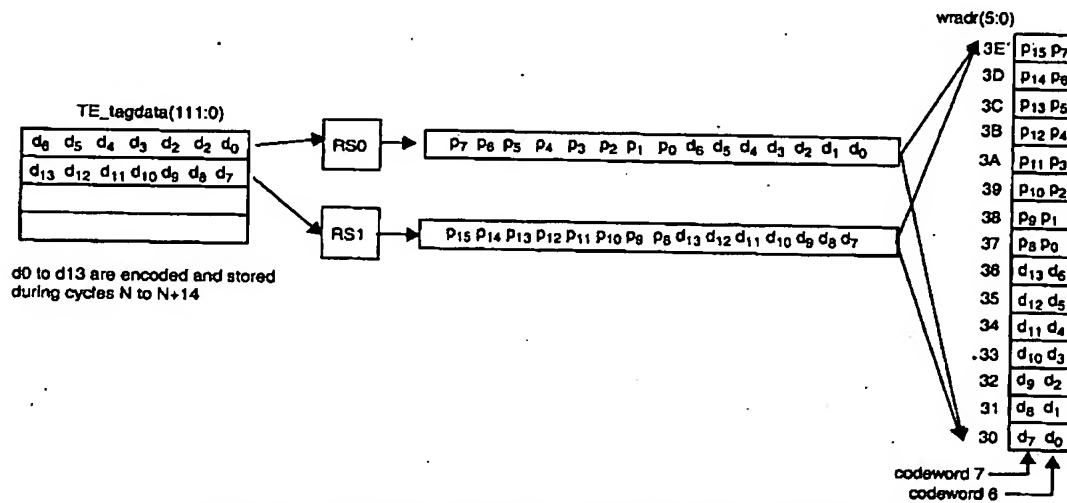
Figure 171. Coding and mapping of uncoded Fixed Tag Data for (15,7) RS encoder

Figure 172. Mapping of 2D decoded Variable Tag Data
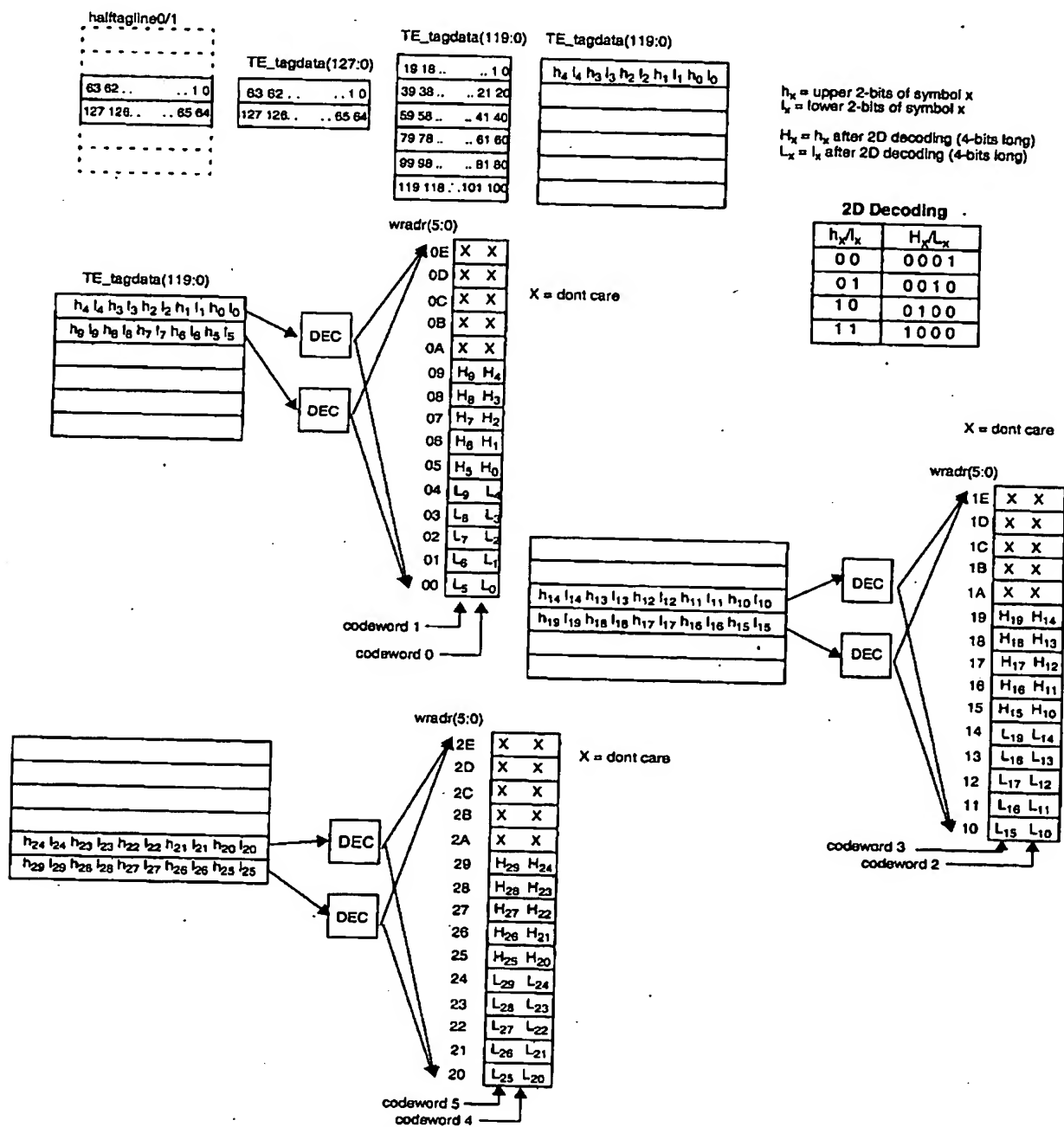
26.7.6 Reed Solomon (RS) Encoder

26.7.7 Introduction

A Reed Solomon code is a non binary, block code. If a symbol consists of m bits then there are $q = 2^m$ possible symbols defining the code alphabet. In the TE, $m = 4$ so the number of possible symbols is $q = 16$.

An (n,k) RS code is a block code with k information symbols and n code-word symbols. RS codes have the property that the code word n is limited to at most $q+1$ symbols in length.

In the case of the TE, both (15,5) and (15,7) RS codes can be used. This means that up to 5 and 4 symbols respectively can be corrected.

Only one type of RS coder is used at any particular time. The RS coder to be used is determined by the registers *TE_dataredun* and *TE_decode2den*:
- *TE_dataredun* = 0 and *TE_decode2den* = 0, then use the (15,5) RS coder
- *TE_dataredun* = 1 and *TE_decode2den* = 0, then use the (15,7) RS coder

For a (15,k) RS code with $m = 4$, k 4-bit information symbols applied to the coder produce 15 4-bit codeword symbols at the output. In the TE, the code is systematic so the first k codeword symbols are the same the as the k input information symbols.

A simple block diagram can be seen in.
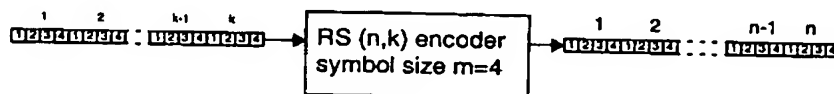
Figure 173. Simple block diagram for an m=4 Reed Solomon Encoder

26.7.8 I/O Specification

A I/O diagram of the RS encoder can be seen in.
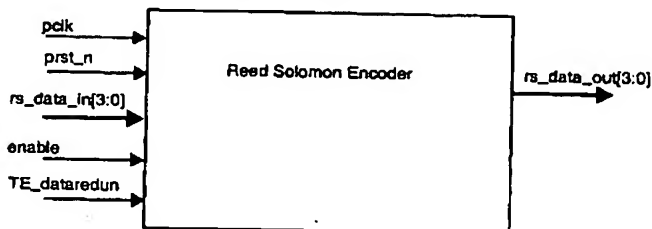
Figure 174. RS Encoder I/O diagram

26.7.9 Proposed implementation

In the case of the TE, (15,5) and (15,7) codes are to be used with 4-bits per symbol.

The primitive polynomial is $p(x) = x^4 + x + 1$

In the case of the (15,5) code, this gives a generator polynomial of

$$g(x) = (x+a)(x+a^2)(x+a^3)(x+a^4)(x+a^5)(x+a^6)(x+a^7)(x+a^8)(x+a^9)(x+a^{10})$$

$$g(x) = x^{10} + a^2 x^9 + a^3 x^8 + a^9 x^7 + a^6 x^6 + a^{14} x^5 + a^2 x^4 + ax^3 + a^6 x^2 + ax + a^{10}$$

$$g(x) = x^{10} + g_9 x^9 + g_8 x^8 + g_7 x^7 + g_6 x^6 + g_5 x^5 + g_4 x^4 + g_3 x^3 + g_2 x^2 + g_1 x + g_0$$

In the case of the (15,7) code, this gives a generator polynomial of

$$h(x) = (x+a)(x+a^2)(x+a^3)(x+a^4)(x+a^5)(x+a^6)(x+a^7)(x+a^8)$$

$$h(x) = x^8 + a^{14}x^7 + a^2x^6 + a^4x^5 + a^2x^4 + a^{13}x^3 + a^5x^2 + a^{11}x + a^6$$

$$h(x) = x^8 + h_7x^7 + h_6x^6 + h_5x^5 + h_4x^4 + h_3x^3 + h_2x^2 + h_1x + h_0$$

The output code words are produced by dividing the generator polynomial into a polynomial made up from the input symbols.

This division is accomplished using the circuit shown in Figure 175.
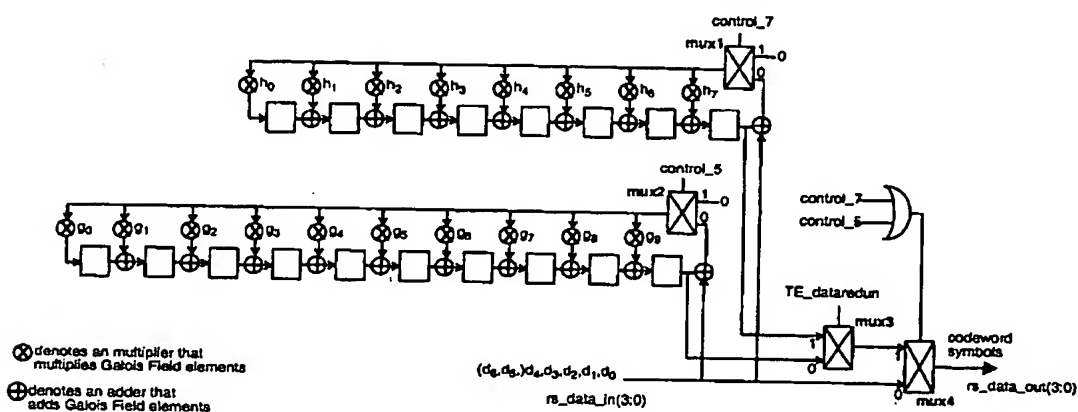
Figure 175. (15,5) & (15,7) RS Encoder block diagram

The data in the circuit are Galois Field elements so addition and multiplication are performed using special circuitry. These are explained in the next sections.

The RS coder can operate either in (15,5) or (15,7) mode. The selection is made by the registers *TE_dataredun* and *TE_decode2den*.

When operating in (15,5) mode *control_7* is always zero and when operating in (15,7) mode *control_5* is always zero.

Firstly consider (15,5) mode i.e. *TE_dataredun* is set to zero.

For each new set of 5 input symbols, processing is as follows:

The 4-bits of the first symbol d_0 are fed to the input port *rs_data_in*(3:0) and *control_5* is set to 0. *mux2* is set so as to use the output as feedback. *control_5* is zero so *mux4* selects the input (*rs_data_in*) as the output (*rs_data_out*). Once the data has settled (<< 1 cycle), the shift registers are clocked. The next symbol d_1 is then applied to the input, and again after the data has settled the shift registers are clocked again. This is repeated for the next 3 symbols d_2, d_3 and d_4. As a result, the first 5 outputs are the same as the inputs. After 5 cycles, the shift registers now contain the next 10 required outputs. *control_5* is set to 1 for the next 10 cycles so that zeros are fed back by *mux2* and the shift register values are fed to the output by *mux3* and *mux4* by simply clocking the registers.

A timing diagram is shown below.
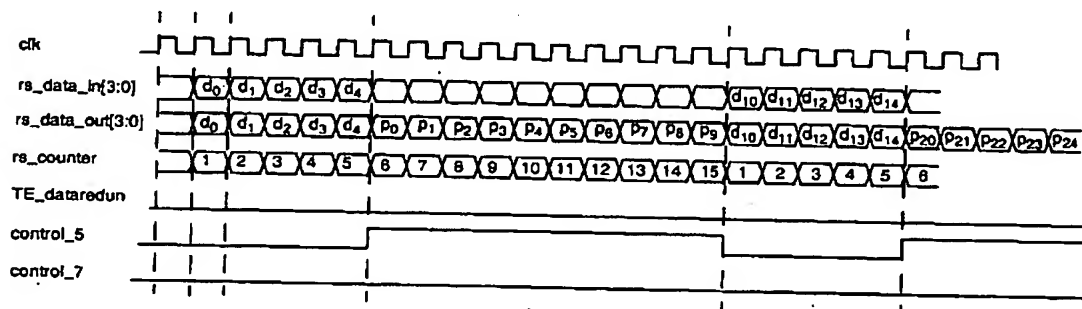
Figure 176. (15,5) RS Encoder timing diagram

Secondly consider (15,7) mode i.e. *TE_dataredun* is set to one.

In this case processing is similar to above except that *control_7* stays low while 7 symbols (d_0, d_1 ... d_6) are fed in. As well as being fed back into the circuit, these symbols are fed to the output. After these 7 cycles, *control_7* is set to 1 and the contents of the shift registers are fed to the output.
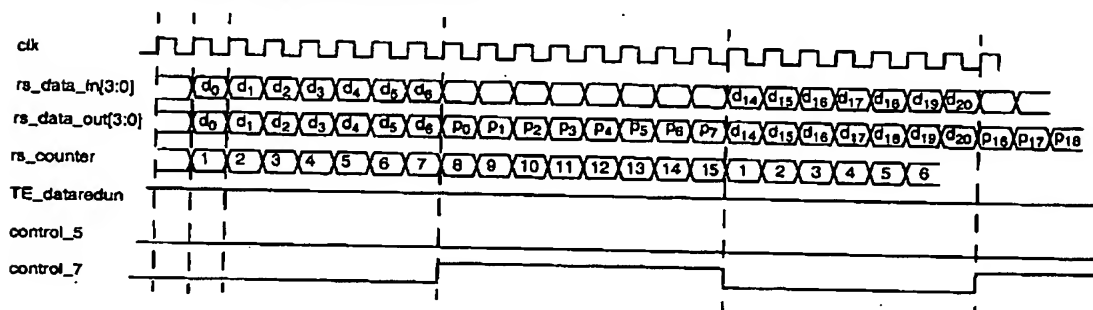
A timing diagram is shown below.

Figure 177. (15,7) RS Encoder timing diagram

The *enable* signal can be used to start/reset the counter and the shift registers.

The RS encoders can be designed so that encoding starts on a rising *enable* edge. After 15 symbols have been output, the encoder stops until a rising *enable* edge is detected. As a result there will be a delay between each codeword.

Alternatively, once the enable goes high the shift registers are reset and encoding will proceed until it is told to stop. *rs_data_in* must be supplied at the correct time. Using this method, data can be continuously output at a rate of 1 symbol per cycle, even over a few codewords.

Alternatively, the RS encoder can request data as it requires.

The performance criterion that must be met is that the following must be carried out within 63 cycles
- load one tag's raw data into *TE_tagdata*
- encode the raw tag data
- store the encoded tag data in the Encoded Tag Data Interface

In the case of the raw fixed tag data at the start of a page, there is no definite performance criterion except that it should be encoded and stored as fast as possible.

26.7.10   Galois Field elements and their representation

A Galois Field is a set of elements in which we can do addition, subtraction, multiplication and division without leaving the set.

The TE uses RS encoding over the Galois Field $GF(2^4)$. There are 2^4 elements in $GF(2^4)$ and they are generated using the primitive polynomial $p(x) = x^4 + x + 1$.

The 16 elements of $GF(2^4)$ can be represented in a number of different ways. Table shows three possible representations - the power, polynomial and 4-tuple representation.

Table 129. GF(2^4) representations

Power representation	Polynomial Representation	4-tuple representation ($a_0 a_1 a_2 a_3$)
0	0	(0 0 0 0)
1	1	(1 0 0 0)
a	X	(0 1 0 0)
α^2	X^2	(0 0 1 0)
α^3	X^3	(0 0 0 1)
α^4	$1 + X$	(1 1 0 0)
α^5	$X + X^2$	(0 1 1 0)
a^6	$X^2 + X^3$	(0 0 1 1)
α^7	$1 + X \quad + X^3$	(1 1 0 1)
α^8	$1 \quad + X^2$	(1 0 1 0)
α^9	$X \quad + X^3$	(0 1 0 1)
α^{10}	$1 + X + X^2$	(1 1 1 0)
α^{11}	$X + X^2 + X^3$	(0 1 1 1)
α^{12}	$1 + X + X^2 + X^3$	(1 1 1 1)
α^{13}	$1 \quad + X^2 + X^3$	(1 0 1 1)
α^{14}	$1 \quad + X^3$	(1 0 0 1)

26.7.11   Multiplication of GF(2^4) elements

The multiplication of two field elements α^a and α^b is defined as

$$\alpha^c = \alpha^a.\alpha^b = \alpha^{(a+b) \bmod 15}$$

Thus

$$\alpha^1.\alpha^2 = \alpha^3$$
$$\alpha^5.\alpha^{10} = \alpha^{15}$$
$$\alpha^6.\alpha^{12} = \alpha^3$$

So if we have the elements in exponential form, multiplication is simply a matter of modulo 15 addition.

If the elements are in polynomial/tuple form, the polynomials must be multiplied and reduced mod $x^4 + x + 1$.

Suppose we wish to multiply the two field elements in $GF(2^4)$:

$$\alpha^a = a_3 x^3 + a_2 x^2 + a_1 x^1 + a_0$$
$$\alpha^b = b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$$

where a_i, b_i are in the field (0,1) (i.e. modulo 2 arithmetic)

Multiplying these out and using $x^4 + x + 1 = 0$ we get:

$$\alpha^{a+b} = [(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0) + a_3b_3]x^3$$
$$+ [(a_0b_2 + a_1b_1 + a_2b_0) + a_3b_3 + (a_3b_2 + a_2b_3)]x^2$$
$$+ [(a_0b_1 + a_1b_0) + (a_3b_2 + a_2b_3) + (a_1b_3 + a_2b_2 + a_3b_1)]x$$
$$+ [(a_0b_0 + a_1b_3 + a_2b_2 + a_3b_1)]$$
$$\alpha^{a+b} = [a_0b_3 + a_1b_2 + a_2b_1 + a_3(b_0 + b_3)]x^3$$
$$+ [a_0b_2 + a_1b_1 + a_2(b_0 + b_3) + a_3(b_2 + b_3)]x^2$$
$$+ [a_0b_1 + a_1(b_0 + b_3) + a_2(b_2 + b_3) + a_3(b_1 + b_2)]x$$
$$+ [a_0b_0 + a_1b_3 + a_2b_2 + a_3b_1]$$

If we wish to multiply an arbitrary field element by a fixed field element we get a more simple form. Suppose we wish to multiply α^b by α^3.

In this case $\alpha^3 = x^3$ so (a0 a1 a2 a3) = (0 0 0 1). Substituting this into the above equation gives

$$\alpha^c = (b_0 + b_3)x^3 + (b_2 + b_3)x^2 + (b_1 + b_2)x + b_1$$

This can be implemented using simple XOR gates as shown in Figure 178

⊕ exclusive OR gate
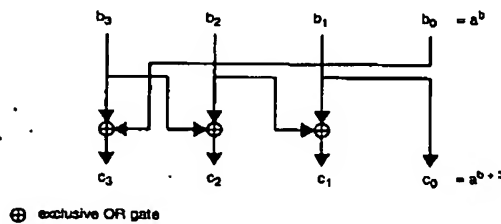
Figure 178. Circuit for multiplying by α^3

26.7.12 Addition of GF(2^4) elements

If the elements are in their polynomial/tuple form, polynomials are simply added.

Suppose we wish to add the two field elements in GF(2^4):

$$\alpha^a = a_3x^3 + a_2x^2 + a_1x + a_0$$
$$\alpha^b = b_3x^3 + b_2x^2 + b_1x + b_0$$

where a_i, b_i are in the field (0,1) (i.e. modulo 2 arithmetic)

$$\alpha^c = \alpha^a + \alpha^b = (a_3 + b_3)x^3 + (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0)$$

Again this can be implemented using simple XOR gates as shown in Figure 179

⊕ exclusive OR gate
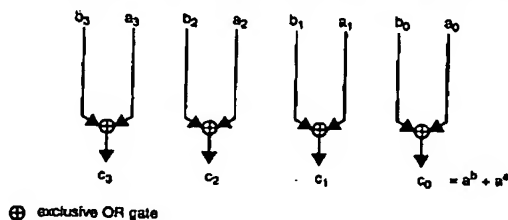
Figure 179. Adding two field elements

26.7.13 Reed Solomon Implementation

The designer can decide to create the relevant addition and multiplication circuits and instantiate them where necessary. Alternatively the feedback multiplications can be combined as follows.

Consider the multiplication

$$\alpha^a.\alpha^b = \alpha^c$$

or in terms of polynomials

$$(a_3x^3 + a_2x^2 + a_1x + a_0).(b_3x^3 + b_2x^2 + b_1x + b_0) = (c_3x^3 + c_2x^2 + c_1x + c_0)$$

If we substitute all of the possible field elements in for α^a and express α^c in terms of α^b, we get the table of results shown in Table 130.

Table 130. α^c multiplied by all field elements, expressed in terms of α^b

fixed field element	$(a_3 a_2 a_1 a_0)$	c_3	c_2	c_1	c_0
0	(0 0 0 0)				
1	(1 0 0 0)	b_0	b_1	b_2	b_3
a	(0 1 0 0)	b_3	b_0+b_3	b_1	b_2
α^2	(0 0 1 0)	b_2	b_2+b_3	b_0+b_3	b_1
α^3	(0 0 0 1)	b_1	b_1+b_2	b_2+b_3	b_0+b_3
α^4	(1 1 0 0)	b_0+b_3	$b_0+b_1+b_3$	b_1+b_2	b_2+b_3
α^5	(0 1 1 0)	b_2+b_3	b_0+b_2	$b_0+b_1+b_3$	b_1+b_2
a^6	(0 0 1 1)	b_1+b_2	b_1+b_3	b_0+b_2	$b_0+b_1+b_3$
α^7	(1 1 0 1)	$b_0+b_1+b_3$	$b_0+b_2+b_3$	b_1+b_3	b_0+b_2
α^8	(1 0 1 0)	b_0+b_2	$b_1+b_2+b_3$	$b_0+b_2+b_3$	b_1+b_3
α^9	(0 1 0 1)	b_1+b_3	$b_0+b_1+b_2+b_3$	$b_1+b_2+b_3$	$b_0+b_2+b_3$
α^{10}	(1 1 1 0)	$b_0+b_2+b_3$	$b_0+b_1+b_2$	$b_0+b_1+b_2+b_3$	$b_1+b_2+b_3$
α^{11}	(0 1 1 1)	$b_1+b_2+b_3$	b_0+b_1	$b_0+b_1+b_2$	$b_0+b_1+b_2+b_3$
α^{12}	(1 1 1 1)	$b_0+b_1+b_2+b_3$	b_0	b_0+b_1	$b_0+b_1+b_2$
α^{13}	(1 0 1 1)	$b_0+b_1+b_2$	b_3	b_0	b_0+b_1
α^{14}	(1 0 0 1)	b_0+b_1	b_2	b_3	b_0

the following signals are required:

* $b_0, b_1, b_2, b_3,$

- (b_0+b_1), (b_0+b_2), (b_0+b_3), (b_1+b_2), (b_1+b_3), (b_2+b_3),
- $(b_0+b_1+b_2)$, $(b_0+b_1+b_3)$, $(b_0+b_2+b_3)$, $(b_1+b_2+b_3)$,
- $(b_0+b_1+b_2+b_3)$

The implementation of the circuit can be seen in Figure . The main components are XOR gates, 4-bit shift registers and multiplexers.

The RS encoder has 4 input lines labelled 0,1,2 & 3 and 4 output lines labelled 0,1,2 & 3. This labelling corresponds to the subscripts of the polynomial/4-tuple representation. The mapping of 4-bit symbols from the TE_tagdata register into the RS is as follows:

- the LSB in the TE_tagdata is fed into line0

- the next most significant LSB is fed into line1

- the next most significant LSB is fed into line2

- the MSB is fed into line3

The RS output mapping to the Encoded tag data interface is similiar. Two encoded symbols are stored in an 8-bit address. Within these 8 bits:

- line0 is fed into the LSB (bit 0/4)

- line1 is fed into the next most significant LSB (bit 1/5)

- line2 is fed into the next most significant LSB (bit 2/6)

- line3 is fed into the MSB (bit 3/7)
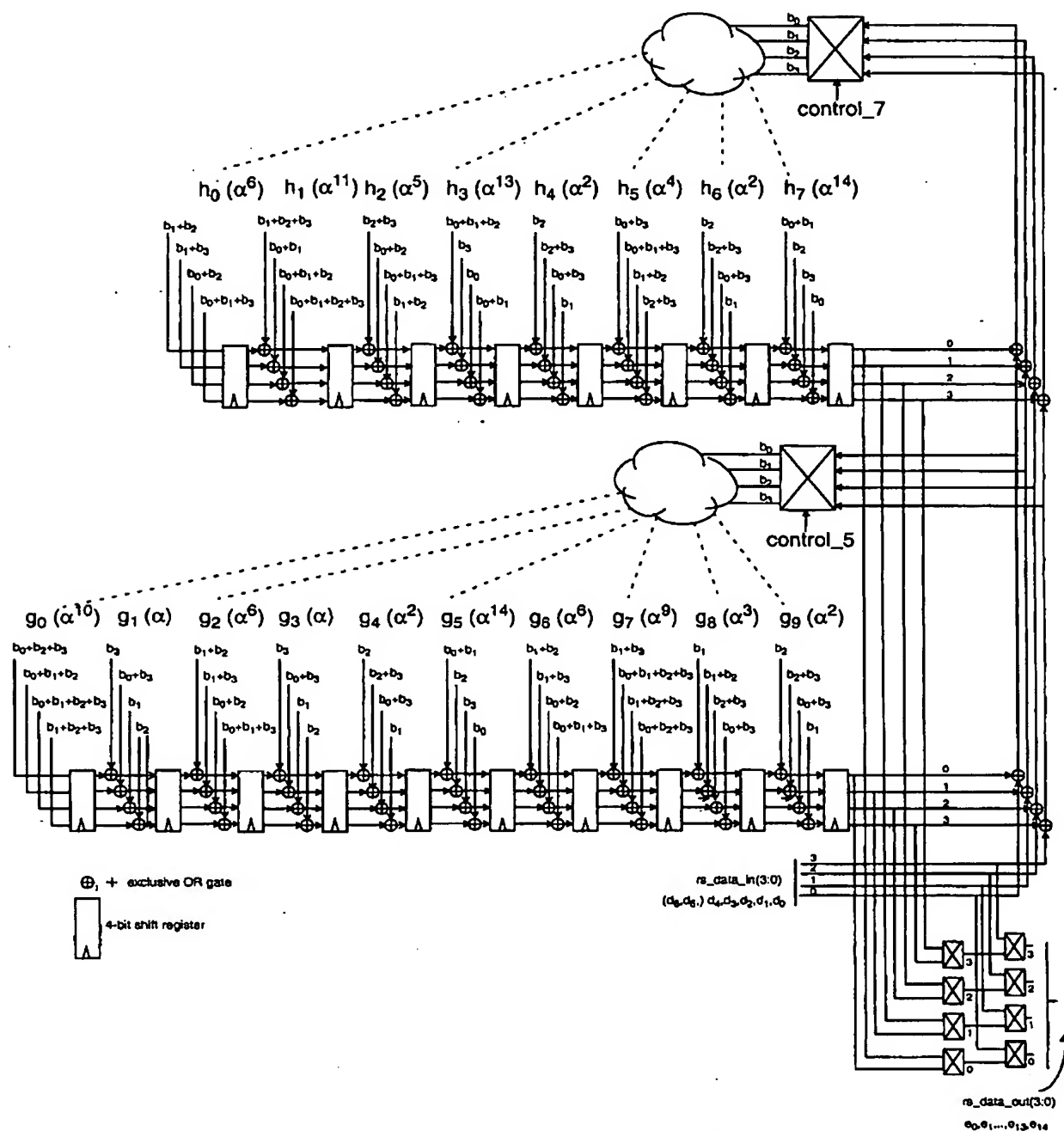
Figure 180. RS Encoder Implementation

26.7.14 2D Decoder

The 2D decoder is selected when TE_decode2den = 1. It operates on variable tag data only. its function is to convert 2-bits into 4-bits according to Table 131.

Table 131. Operation of 2D decoder

Input	Output
0 0	0 0 0 1
0 1	0 0 1 0
1 0	0 1 0 0
1 1	1 0 0 0

26.7.15 Encoded tag data interface

The encoded tag data interface contains an encoded fixed tag data store interface and an encoded variable tag data store interface, as shown in Figure 181.
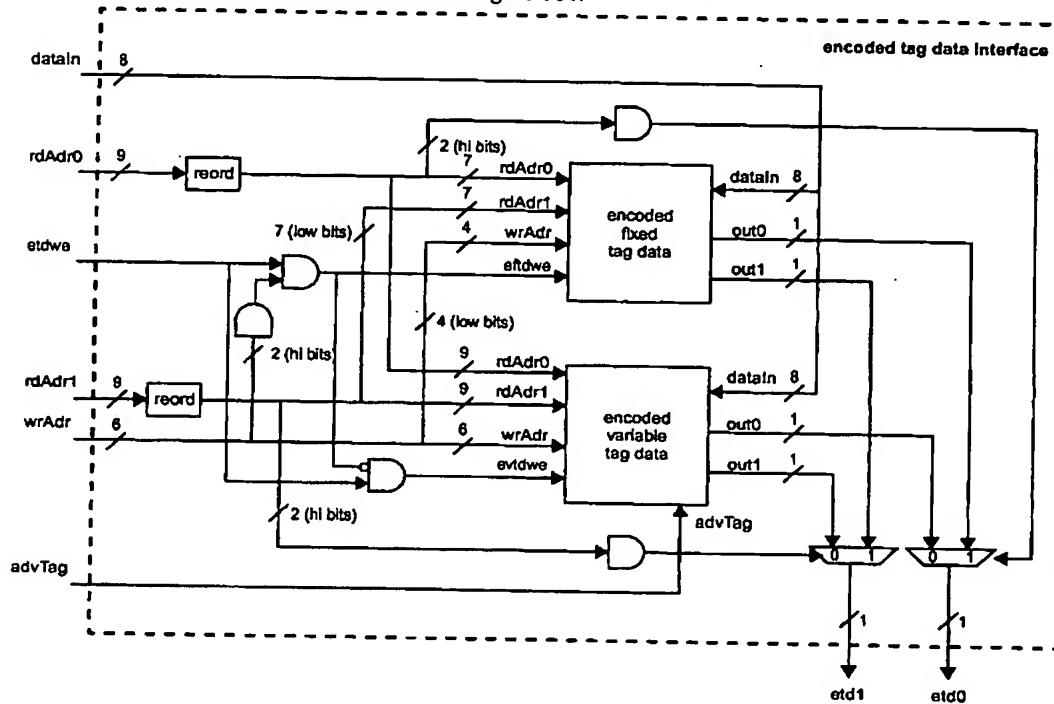
Figure 181. encoded tag data interface

The two reord units simply reorder the 9 input bits to map low-order codewords into the bit selection component of the address as shown in Table 132. Reordering of write addresses is not necessary since the addresses are already in the correct format.

Table 132. Reord unit

bit	bit	Input interpretation	bit	Output interpretation
8	A		A	select 1 of 4 codeword tables
7	B	select 1 of 8 codewords	B	
6	C		D	
5	D		E	select 1 of 15 symbols
4	E	select 1 of 15 symbols	F	
3	F		G	
2	G		C	
1	H	select 1 of 4 bits	H	select 1 of 8 bits
0	I		I	

The encoded fixed data interface is a single 15×8-bit RAM with 2 read ports and 1 write port. As it is only written to during page setup time (it is fixed for the duration of a page) there is no need for simultaneous read/write access. However the fixed data store must be capable of decoding two simultaneous reads in a single cycle. Figure 182 shows the implementation of the fixed data store.
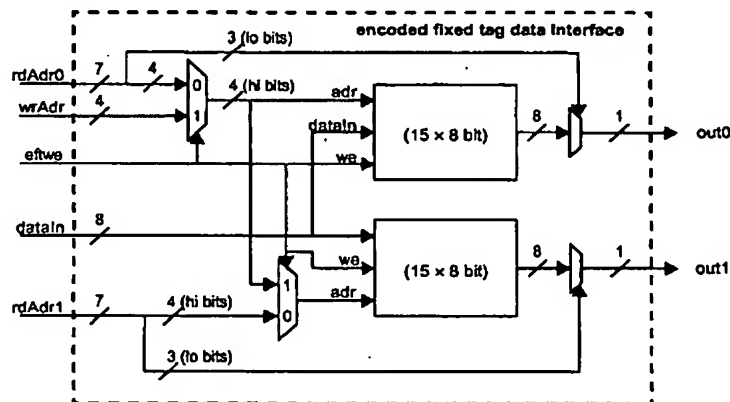
Figure 182. encoded fixed tag data interface

The encoded variable tag data interface is a double buffered $3 \times 15 \times 8$-bit RAM with 2 read ports and 1 write port. The double buffering allows one tag's data to be read (two reads in a single cycle) while the next tag's variable data is being stored. Write addressing is 6 bits: 2 bits of address for selecting 1 of 3, and 4 bits of address for selecting 1 of 15. Read addressing is the same with the addition of 3 more address bits for selecting 1 of 8.

Figure 183 shows the implementation of the encoded variable tag data store. Double buffering is implemented via two sub-buffers. Each time an *AdvTag* pulse is received, the sense of which sub-buffer is being read from or written to changes. This is accomplished by a 1-bit flag called *wrsb0*. Although the initial

state of *wrsb0* is irrelevant, it must invert upon receipt of an *AdvTag* pulse. The structure of each sub-buffer is shown in Figure 184.
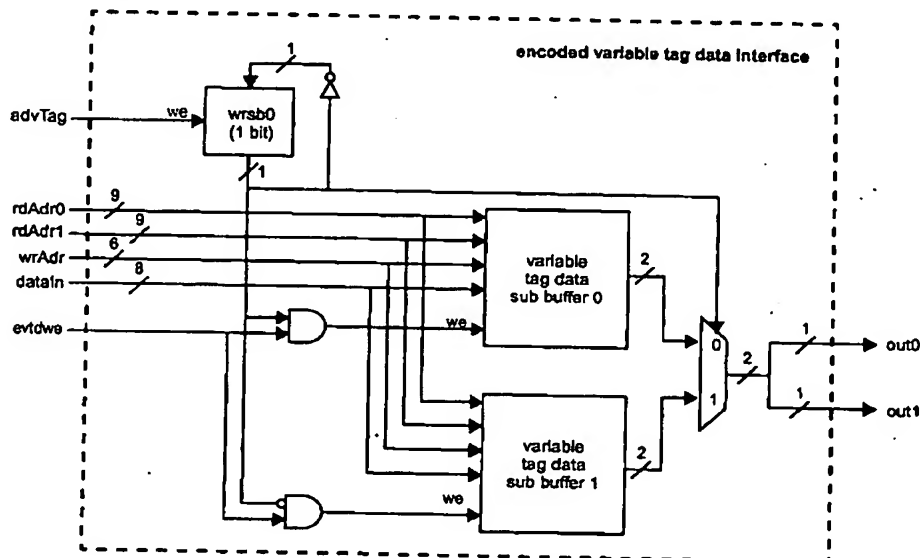
Figure 183. Encoded variable tag data interface
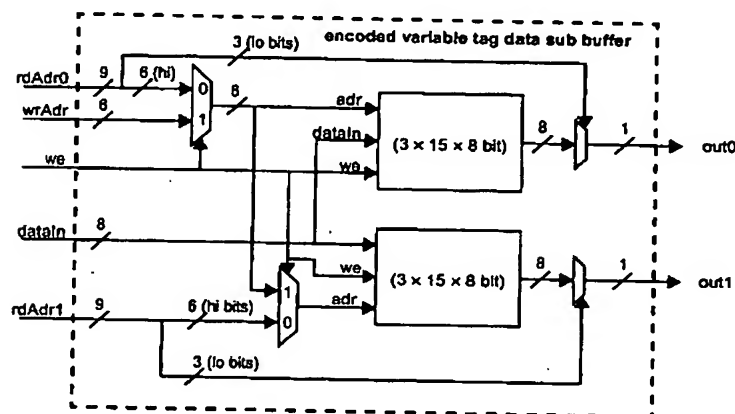
Figure 184. Encoded variable tag data sub-buffer

26.8 TAG FORMAT STRUCTURE (TFS) INTERFACE

26.8.1 Introduction

The TFS specifies the contents of every dot position within a tags border i.e.:
- is the dot part of the background?
- is the dot part of the data?

The TFS is broken up into Tag Line Structures (TLS) which specify the contents of every dot position in a particular line of a tag. Each TLS consists of three tables - A, B and C (see Figure 185).

For a given line of dots, all the tags on that line correspond to the same tag line structure. Consequently, for a given line of output dots, a single tag line structure is required, and not the entire TFS. Double buffering allows the next tag line structure to be fetched from the TFS in DRAM while the existing tag line structure is used to render the current tag line.

The TFS interface is responsible for loading the appropriate line of the tag format structure as the tag encoder advances through the page. It is also responsible for producing table A and table B outputs for two consecutive dot positions in the current tag line.
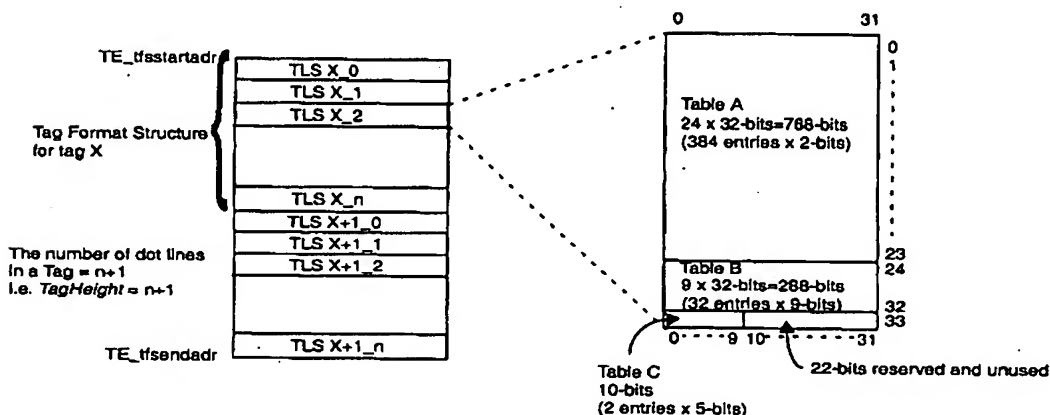
Figure 185. Breakdown of the Tag Format Structure

- There is a TLS for every dot line of a tag.
- All tags that are on the same line have the exact same TLS.
- A tag can be up to 384 dots wide, so each of these 384 dots must be specified in the TLS.
- The TLS information is stored in DRAM and one TLS must be read in to the TFS Interface for each line of dots that are outputted to the Tag Plane Line Buffers.
- Each TLS consists of 17 64-bits words. This is read from DRAM as 5 times 256-bit words with 192 padded bits in the last 256-bit DRAM read.

26.8.2 I/O Specification

Table 133. Tag Format Structure Interface Port List

Signal name	Signal type	description
pclk	In	SoPEC system clock
prst_n	In	Active-low, synchronous reset in pclk domain

Table 133. Tag Format Structure Interface Port List

signal name	signal type	description
top_go	In	Go signal from TE top level
DRAM		
diu_data[63:0]	In	Data from DRAM
diu_tfs_rack	In	Data acknowledge from DRAM
diu_tfs_rvalid	In	Data valid from DRAM
tfs_diu_rreq	Out	Read request to DRAM
tfs_diu_radr[21:5]	Out	Read address to DRAM
tag encoder top level		
top_advtagline	In	Pulsed after the last line of a row of tags
top_tagaltsense	In	For even tag rows = 0 i.e. 0,2,4.. For odd tag rows = 1 i.e. 1,3,5...
top_lastdotintag	In	Last dot in tag is currently being processed
top_dotposvalid	In	Current dot position is a tag dot and its structure data and tag data is available
top_tagdotnum[7:0]	In	Counts from zero up to *TE_tagmaxdotpairs* (min. =1, max. = 192)
tfsi_valid	Out	TLS tables A, B and C, ready for use
tfsi_ta_dot0[1:0]	Out	Even entry from Table A corresponding to top_tagdotnum
tfsi_ta_dot1[1:0]	Out	Odd entry from Table A corresponding to top_tagdotnum
tag encoder top level (PCU read decoder)		
tfs_te_tfsstartadr[23:0]	Out	TFS tfsstartadr register
tfs_te_tfsendadr[23:0]	Out	TFS tfsendadr register
tfs_te_tfsfirstlineadr[23:0]	Out	TFS tfsfirstlineadr register
tfs_te_currtfsadr[23:0]	Out	TFS currtfsadr register
TDI		
tfsi_tdi_adr0[8:0]	Out	Read address for dot0 (even dot)
tfsi_tdi_adr1[8:0]	Out	Read address for dot1 (odd dot)

26.8.2.1 State machine

The state machine is responsible for generating control signals for the various TFS table units, and to load the appropriate line from the TFS. The states are explained below.

idle:- Wait for top_go to become active. Pulse *adv_tfs_line* for 1 cycle to reset *tawradr* and *tbwradr* registers. Pulsing *adv_tfs_line* will switch the read/write sense of Table B so switching Table A here as well to keep things the same i.e. *wrta0* = NOT(*wrta0*).

diu_access:- In the diu_access state a request is sent to the DIU. Once an *ack* signal is received Table A write enable is asserted and the FSM moves to the tls_load state.

tls_load:- The DRAM access is a burst of 5 256-bit accesses, ultimately returned by the DIU as 5*(4*64bit) words. There will be 192 padded bits in the last 256-bit DRAM word. The first 12 64-bit words reads are for Table A, words 12 to 15 and some of 16 are for Table B while part of read 16 data is for Table C. The counter *read_num* is used to identify which data goes to which table. The table B data is stored temporarily in a 288-bit register until the tls_update state hence *tbwe* does not become active until read_num = 16).
- The DIU data goes directly into Table A (12 * 64).
- The DIU data for Table B is loaded into a 288-bit register.
- The DIU data goes directly into Table C.

tls_update:- The 288-bits in Table B need to written to a 32*9 buffer. The tls_update state takes care of this using the *read_num* counter.

tls_next:- This state checks the logic level of *tfsvalid* and switches the read/write senses of Table A (*wrta0*) and Table B a cycle later (using the *adv_tfs_line* pulse). The reason for switching Table A a cycle early is to make sure the top_level address via *tagdotnum* is pointing to the correct buffer. Keep in mind the top_level is working a cycle ahead of Table A and 2 cycles ahead of Table B.

If *tfsValid* is 1, the state machine waits until the *advTagLine* signal is received. When it is received, the state machine pulses *advTFSLine* (to switch read/write sense in tables A, B, C), and starts reading the next line of the TFS from *currTFSAdr*.

If *tfsValid* is 0, the state machine pulses *advTFSLine* (to switch read/write sense in tables A, B, C) and then jumps to the tls_tfsvalid_set state where the signal *tfsValid* is set to 1 (allowing the tag encoder to start, or to continue if it had been stalled). The state machine can then start reading the next line of the TFS from *currTFSAdr*.

tls_tfsvalid_next:- Simply sets the *tfsvalid* signal and returns the FSM to the diu_access state.

If an *advTagLine* signal is received before the next line of the TFS has been read in, *tfsValid* is cleared to 0 and processing continues as outlined above.

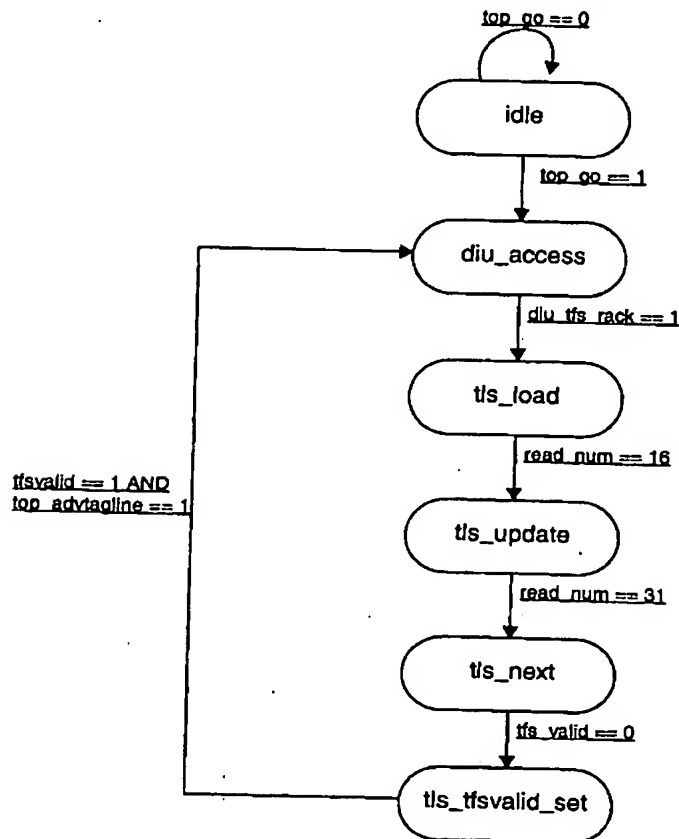The TFS state flow diagram is shown in below..

Figure 186. TFSI FSM State Flow Diagram

26.8.3 Generating a tag from Tables A, B and C

The TFS contains an entry for each dot position within the tag's bounding box. Each entry specifies whether the dot is part of the constant background pattern or part of the tag's data component (both fixed and variable).

The TFS therefore has TagHeight x TagWidth entries, where TagHeight is the height of the tag in dot-lines and TagWidth is the width of the tag in dots. The TFS entries that specify a single dot-line of a tag are known as a Tag Line Structure.

The TFS contains a TLS for each of the 1600 dpi lines in the tag's bounding box. Each TLS contains three contiguous tables, known as tables A, B and C.

Table A contains 384 2-bit entries i.e. one entry for each dot in a single line of a tag up to the maximum width of a tag. The actual number of entries used should match the size of the bounding box for the tag in the dot dimension, but all 384 entries must be present.

Table B contains 32 9-bit data address that refer to (in order of appearance) the data dots present in the particular line. Again, all 32 entries must be present, even if fewer are used.

Table C contains two 5-bit pointers into table B and is followed by 22 unused bits. The total length of each TLS is therefore 34 32-bit words.

Each output dot value is generated as follows: Each entry in Table A consists of 2-bits - bit0 and bit1. These 2-bits are interpreted according to Table , Table and Table .

Table 134. Interpretation of bit0 from entry in Table A

bit0	Interpretation
0	the output bit comes directly from bit1 (see Table).
1	the output bit comes from a data bit. Bit1 is used in conjunction with Tag Line Structure Table B to determine which data bit will be output.

Table 135. Interpretation of bit1 from entry in table A when bit0 = 0

bit1	Interpretation
0	output 0
1	output 1

Table 136. Interpretation of bit1 from entry in table A when bit0 = 1

bit1	Interpretation
0	output data bit pointed to by current index into Table B.
1	output data bit pointed to by current index into Table B, and advance index by 1.

If bit0 = 0 then the output dot for this entry is part of the constant background pattern. The dot value itself comes from bit1 i.e. if bit1 = 0 then the output is 0 and if bit1 = 1 then the output is 1.

If bit0 = 1 then the output dot for this entry comes from the **variable or fixed tag data**. Bit1 is used in conjunction with Tables B and C to determine data bits to use.

To understand the interpretation of bit1 when bit0 = 1 we need to know what is stored in Table B. Table B contains the addresses of all the data bits that are used in the particular line of a tag in order of appearance. Therefore, up to 32 different data bits can appear in a line of a tag. The address of the first data dot in a tag will be given by the address stored in entry 0 of Table B. As we advance along the various data dots we will advance through the various Table B entries.

Each Table B entry is 9-bits long and each points to a specific **variable** or **fixed data** bit for the tag. Each tag contains a maximum of 120 **fixed** and 360 **variable data bits**, for a total of 480 data bits. To aid address decoding, the addresses are based on the RS encoded tag data. Table lists the interpretation of the 9-bit addresses.

Table 137. Interpretation of 9-bit tag data address in Table B

bit pos	name	description
8	CodeWordSelect	Select 1 of 8 codewords.
7		Codewords 0, 1, 2, 3, 4, 5 are variable data.
6		Codewords 6, 7 are fixed data.

Table 137. Interpretation of 9-bit tag data address in Table B

bit pos	name	description
	SymbolSelect	Select 1 of 15 symbols (1111 invalid)
	BitSelect	Select 1 of 4 bits from the selected symbols

If the fixed data is supplied to the TE in an unencoded form, the symbols derived from codeword 0 of fixed data are written to codeword 6 and the symbols derived from fixed data codeword 1 are written to codeword 7. The data symbols are stored first and then the remaining redundancy symbols are stored afterwards, for a total of 15 symbols. Thus, when 5 data symbols are used, the 5 symbols derived from bits 0-19 are written to symbols 0-4, and the redundancy symbols are written to symbols 5-14. When 7 data symbols are used, the 7 symbols derived from bits 0-27 are written to symbols 0-6, and the redundancy symbols are written to symbols 7-14

However, if the fixed data is supplied to the TE in a pre-encoded form, the encoding could theoretically be anything. Consequently the 120 bits of fixed data is copied to codewords 6 and 7 as shown in Table 138.

Table 138. Mapping of fixed data to codeword/symbols when no redundancy encoding

input bits	output symbol range	output codeword
0-19	0-4	6
20-39	0-4	7
40-59	5-9	6
60-79	5-9	7
80-99	10-14	6
100-119	10-14	7

It is important to note that the interpretation of bit1 from Table A (when bit0 = 1) is relative. A 5-bit index is used to cycle through the data address in Table B. Since the first tag on a particular line may or may not start at the first dot in the tag, an initial value for the index into Table B is needed. Subsequent tags on the same line will always start with an index of 0, and any partial tag at the end of a line will simply finish before the entire tag has been rendered. The initial index required due to the rendering of a partial tag at the start of a line is supplied by Table C. The initial index will be different for each TLS and there are two possible initial indexes since there are effectively two types of rows of tags in terms of initial offsets.

Table C provides the appropriate start index into Table B (2 5-bit indices). When rendering even rows of tags, entry 0 is used as the initial index into Table B, and when rendering odd rows of tags, entry 1 is used as the initial index into Table B. The second and subsequent tags start at the left most dots position within the tag, so can use an initial index of 0.

26.8.4 Architecture

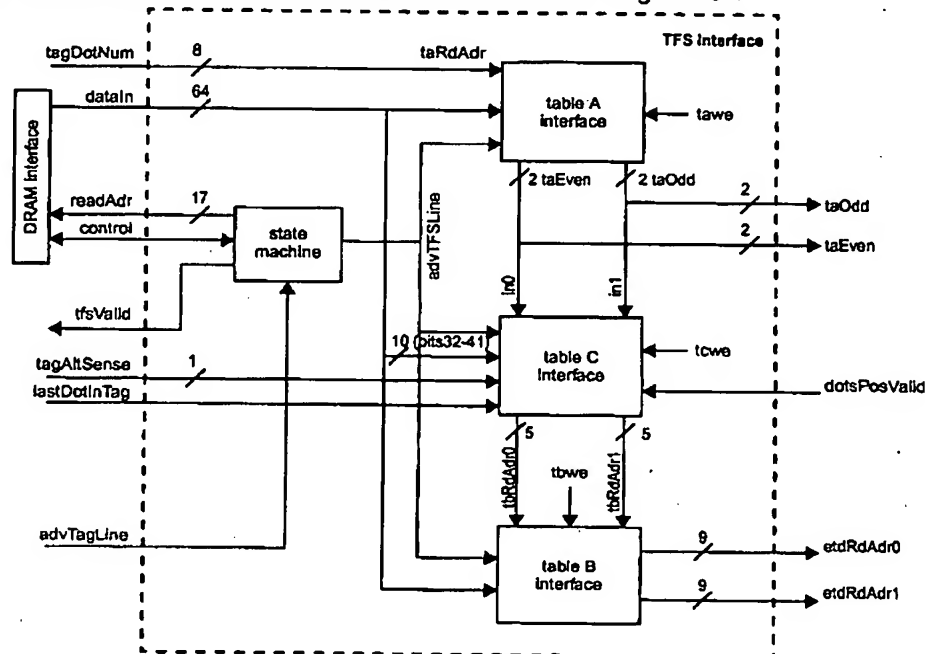A block diagram of the Tag Format Structure Interface can be seen in Figure 187.

Figure 187. TFS Block Diagram

26.8.4.1 Table A Interface

The implementation of table A is two 16 × 64-bit RAMs with a small amount of control logic, as shown in Figure 188. While one RAM is read from for the current line's table A data (4 bits representing 2 contiguous table A entries), the other RAM is being written to with the next line's table A data (64-bits at a time).
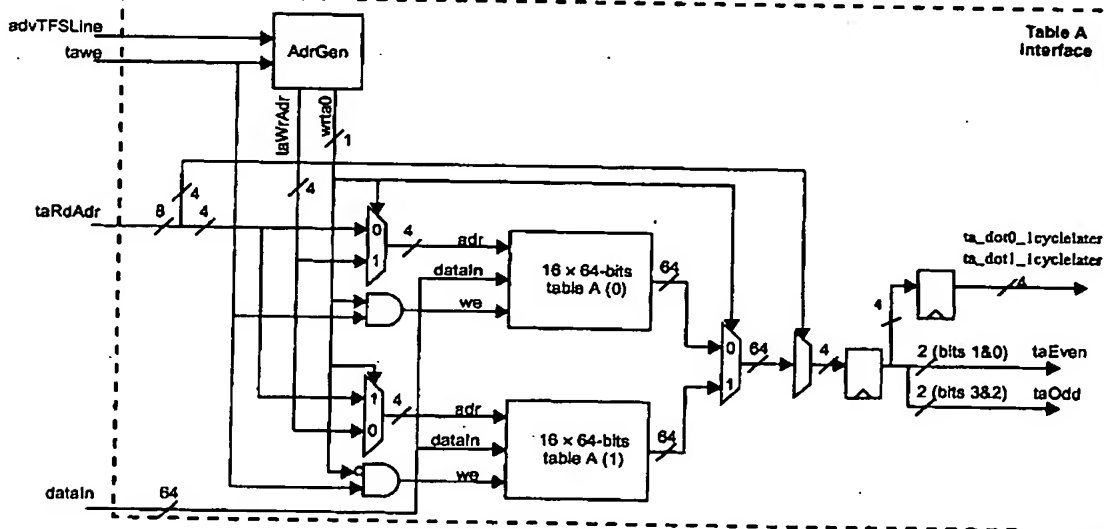

Figure 188. Table A interface block diagram

Note:- The Table A data to be printed (if each LSB = 0) must be passed to the top_level 2 cycles after the read of Table A due to the 2-stage pipelining in the TFS from registering Table A and Table B outputs hence this extra registering stage for the generation of ta_dot0_1cyclelater and ta_dot1_1cyclelater.

Each time an *AdvTFSLine* pulse is received, the sense of which RAM is being read from or written to changes. This is accomplished by a 1-bit flag called *wrta0*. Although the initial state of *wrta0* is irrelevant, it must invert upon receipt of an *AdvTFSLine* pulse. A 4-bit counter called *taWrAdr* keeps the write address for the 12 writes that occur after the start of each line (specified by the *AdvTFSLine* control input). The *tawe* (table A write enable) input is set whenever the data in is to be written to table A. The *taWrAdr* address counter automatically increments with each write to table A. Address generation for *tawe* and *taWrAdr* is shown in Table 189.
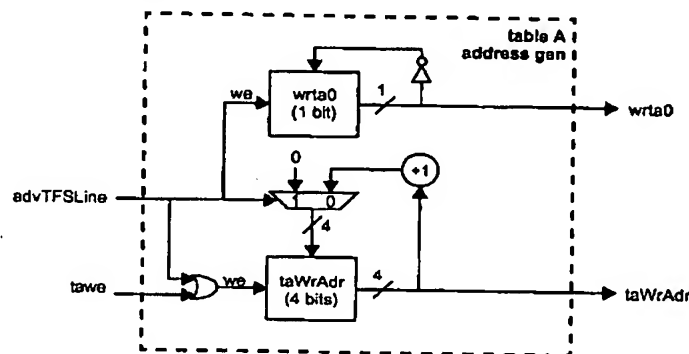
Figure 189. Table A address generator

26.8.4.2 Table C Interface

A block diagram of the table C interface is shown below in Figure 190.
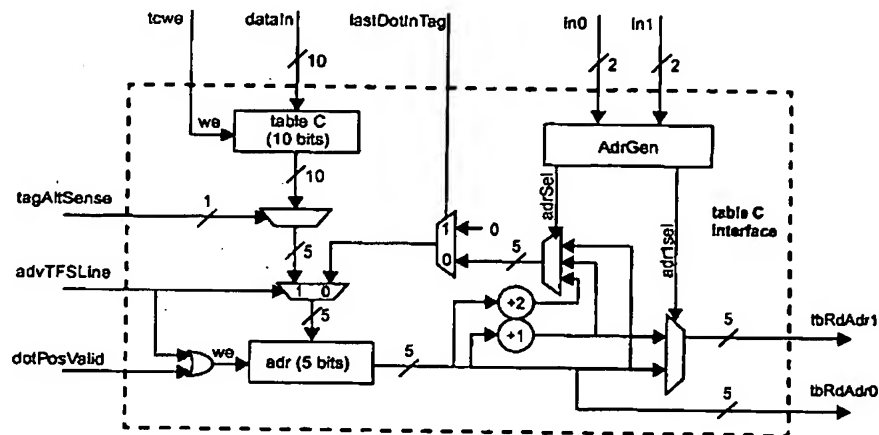
Figure 190. Table C interface block diagram

The address generator for table C contains a 5 bit address register *adr* that is set to a new address at the start of processing the tag (either of the two table C initial values based on *tagAltSense* at the start of the line, and 0 for subsequent tags on the same line). Each cycle two addresses into table B are generated based on the two 2-bit inputs (*in0* and *in1*). As shown inSection 139, the output address *tbRdAdr0* is always *adr* and *tbRdAdr1* is one of *adr* and *adr+1*, and at the end of the cycle *adr* takes on one of *adr*, *adr+1*, and *adr+2*.

Table 139. AdrGen lookup table

inputs		outputs		
in0	in1	adr0Sel	adr1Sel	adrSel
00	00	X[1]	X	adr
00	01	X	adr	adr
00	10	X	X	adr
00	11	X	adr	adr+1
01	00	adr	X	adr
01	01	adr	adr	adr
01	10	adr	X	adr
01	11	adr	adr	adr+1
10	00	X	X	adr
10	01	X	adr	adr
10	10	X	X	adr
10	11	X	adr	adr+1
11	00	adr	X	adr+1
11	01	adr	adr+1	adr+1
11	10	adr	X	adr+1
11	11	adr	adr+1	adr+2

1. X = don't care state.

26.8.4.3 Table B interface

The table B interface implementation generates two encoded tag data addresses (*tfsi_adr0*, *tfsi_adr1*) based on two table B input addresses (*tbRdAdr0*, *tbRdAdr1*). A block diagram of table B can be seen in Figure 191.
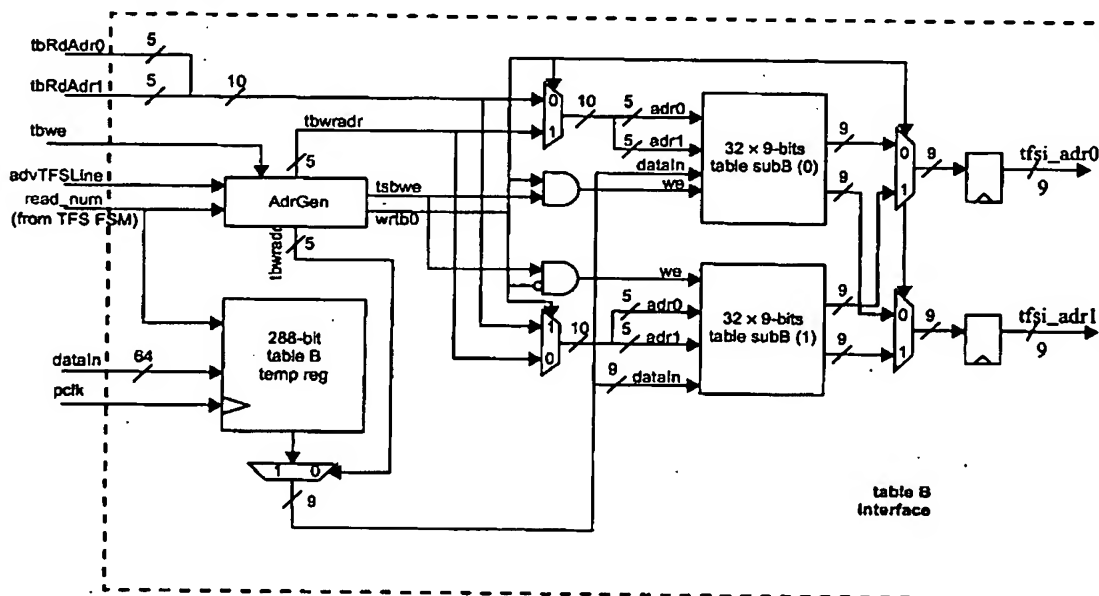
Figure 191. Table B Interface block diagram

Table B data is initially loaded into the 288-bit table B temporary register via the TFS FSM. Once all 288-bit entries have been loaded from DRAM, the data is written in 9-bit chunks to the 32*9 register arrays based on tbwradr.

Each time an *AdvTFSLine* pulse is received, the sense of which sub buffer is being read from or written to changes. This is accomplished by a 1-bit flag called *wrtb0*. Although the initial state of *wrtb0* is irrelevant, it must invert upon receipt of an *AdvTFSLine* pulse.

' Note:- The output addresses from Table B are registered.

27 Tag FIFO Unit (TFU)

27.1 OVERVIEW

The Tag FIFO Unit (TFU) provides the means by which data is transferred between the Tag Encoder (TE) and the HCU. By abstracting the buffering mechanism and controls from both units, the interface is clean between the data user and the data generator.

The TFU is a simple FIFO interface to the HCU. The Tag Encoder will provide support for arbitrary Y integer scaling up to 1600 dpi. X integer scaling of the tag dot data is performed at the output of the FIFO in the TFU. There is feedback to the TE from the TFU to allow stalling of the TE during a line. The TE interfaces to the TFU with a data width of 8 bits. The TFU interfaces to the HCU with a data width of 1 bit.

The depth of the TFU FIFO is chosen as 16 bytes so that the FIFO can store a single 126 dot tag.

27.1.1 Interfaces between TE, TFU and HCU
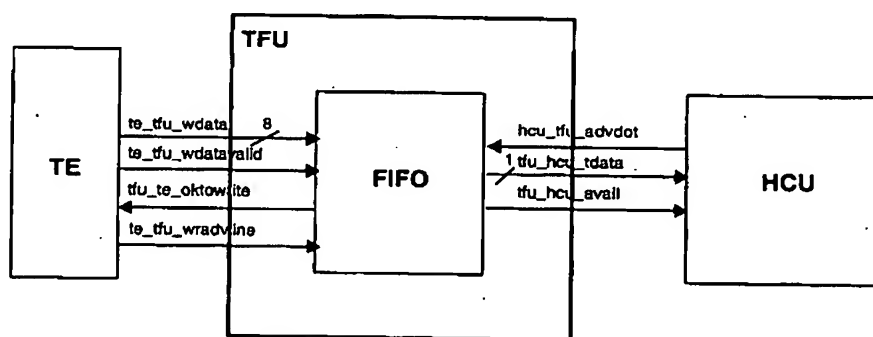
Figure 192. Interfaces between TE, TFU and HCU

27.1.1.1 TE-TFU Interface

The interface from the TE to the TFU comprises the following signals:
- *te_tfu_wdata*, 8-bit write data.
- *te_tfu_wdatavalid*, write data valid.
- *te_tfu_wradvline*, accompanies the last valid 8-bit write data in a line.

The interface from the TFU to TE comprises the following signal:
- *tfu_te_oktowrite*, indicating to the TE that there is space available in the TFU FIFO.

The TE writes data to the TFU FIFO as long as the TFU's *tfu_te_oktowrite* output bit is set. The TE write will not occur unless data is accompanied by a data valid signal.

27.1.1.2 TFU-HCU Interface

The interface from the TFU to the HCU comprises the following signals:
- *tfu_hcu_tdata*, 1-bit data.
- *tfu_hcu_avail*, data valid signal indicating that there is data available in the TFU FIFO.

The interface from HCU to TFU comprises the following signal:
- *hcu_tfu_ready*, indicating to the TFU to supply the next dot.

27.1.1.2.1 X scaling

Tag data is replicated a scale factor (SF) number of times in the X direction to convert the final output to 1600 dpi. Unlike both the CFU and SFU, which support non-integer scaling, the scaling is integer only. Replication in the X direction is performed at the output of the TFU FIFO on a dot-by-dot basis.

To account for the case where there may be two SoPEC devices, each generating its own portion of a dot-line, the first dot in a line may not be replicated the total scale-factor number of times by an individual TFU. The dot will ultimately be scaled-up correctly with both devices doing part of the scaling, one on its lead-out and the other on its lead in.

Note two SoPEC TEs may be involved in producing the same byte of output tag data straddling the print-head boundary. The HCU of the left SoPEC will accept from its TE the correct amount of dots, ignoring any dots in the last byte that do not apply to its printhead. The TE of the right SoPEC will be programmed the correct number of dots into the tag and its output will be byte aligned with the left edge of the print-head.

SoPEC : Hardware Design

27.2 DEFINITIONS OF I/O

Table 140. TFU Port List

Port Name	#Pins	I/O	Description
Clocks and Resets			
pclk	1	In	SoPEC Functional clock.
prst_n	1	In	Global reset signal.
PCU interface data and control signals			
pcu_addr[3:2]	2	In	PCU address bus. Only 2 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
tfu_pcu_datain[31:0]	32	Out	Read data bus from the TFU to the PCU.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_tfu_sel	1	In	Block select from the PCU. When pcu_tfu_sel is high both pcu_addr and pcu_dataout are valid.
tfu_pcu_rdy	1	Out	Ready signal to the PCU. When tfu_pcu_rdy is high it indicates the last cycle of the access. For a write cycle this means pcu_dataout has been registered by the block and for a read cycle this means the data on tfu_pcu_datain is valid.
TE interface data and control signals			
te_tfu_wdata[7:0]	8	In	Write data for TFU FIFO.
te_tfu_wdatavalid	1	In	Write data valid signal.
te_tfu_wradvline	1	In	Advance line signal strobed when the last byte in a line is placed on te_tfu_wdata
tfu_te_oktowrite	1	Out	Ready signal indicating TFU has space available in it's FIFO and is ready to be written to.
HCU interface data and control signals			
hcu_tfu_advdot	1	In	Signal indicating to the TFU that the HCU is ready to accept the next dot of data from TFU.
tfu_hcu_tdata	1	Out	Data from the TFU FIFO.
tfu_hcu_avail	1	Out	Signal indicating valid data available from TFU FIFO.

27.3 CONFIGURATION REGISTERS

Table 141. TFU Configuration Registers

Address TFU_Base+	register name	#bits	value on reset	description
Control registers				
0x00	Reset	1	1	A write to this register causes a reset of the SFU. This register can be read to indicate the reset state: 0 - reset in progress 1 - reset not in progress.

Table 141. TFU Configuration Registers

Address TFU_Base +	register name	bits	value on reset	description
0x04	Go	1	see text	Writing 1 to this register starts the TFU. Writing 0 to this register halts the TFU. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. When *Go* is asserted all counters are reset, but configuration registers keep their values (i.e. they don't get reset). The TFU must be started before the TE is started. This register can be read to determine if the TFU is running (1 = running, 0 = stopped).
Setup registers (constant during processing of page)				
0x08	XScale	8	1	Tag scale factor in X direction.
0x0C	XFracScale	8	1	Tag scale factor in X direction for the first dot in a line
0x10	TEByteCount	12	0	The number of bytes to be accepted from the TE per line. Once this number of bytes have been received subsequent bytes are ignored until there is a strobe on the *te_tfu_wradvline*
0x14	HCUDotCount	15	0	The number of (optionally) x-scaled dots per line to be supplied to the HCU. Once this number has been reached the remainder of the current FIFO byte is ignored.

27.4 DETAILED DESCRIPTION

The FIFO is a simple 16-byte store with read and write pointers, and a contents store, Figure 193. 16 bytes is sufficient to store a single 126 dot tag.

Each line a total of *TEByteCount* bytes is read into the FIFO. All subsequent bytes are ignored until there is a strobe on the *te_tfu_wradvline* signal, whereupon bytes for the next line are stored.

On the HCU side, a total of *HCUDotCount* dots are produced at the output. Once this count is reached any more dots in the FIFO byte currently being processed are ignored. For the first dot in the next line the start of line scale factor, *XFracScale*, is used.

The behaviour of these signals and the control signals between the TFU and the TE and HCU is detailed below.
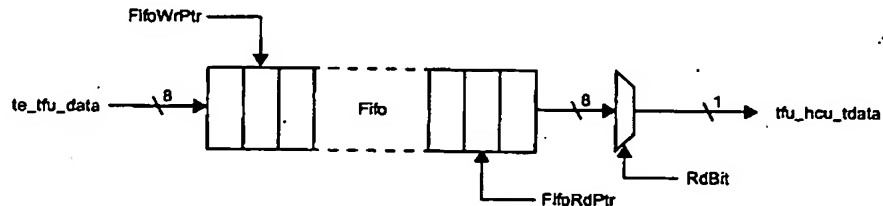

Figure 193. 16-byte FIFO in TFU

```
// Concurrently Executed Code:
// TE always allowed to write when there's either (a) room or (b) no room and all
// bytes for that line have been received.
if ((FifoCntnts != FifoMax) OR (FifoCntnts == FifoMax and ByteToRx == 0)) then
    tfu_te_oktowrite = 1
else
    tfu_te_oktowrite = 0


// Data presented to HCU when there is (a) data in FIFO and (b) the HCU has not
// received all dots for a line
if (FifoCntnts != 0) AND (BitToTx != 0)then
    tfu_hcu_avail = 1
else
    tfu_hcu_avail = 0


// Output mux of FIFO data
tfu_hcu_tdata = Fifo[FifoRdPnt][RdBit]


// Sequentially Executed Code:
if (te_tfu_wdatavalid == 1) AND (FifoCntnts != FifoMax) AND (ByteToRx != 0) then
    Fifo[FifoWrPnt] = te_tfu_wdata
    FifoWrPnt ++
    FifoContents ++
    ByteToRx --

if (te_tfu_wradvline == 1) then
    ByteToRx = TEByteCount


if (hcu_tfu_advdot == 1 and FifoCntnts != 0) then {
    BitToTx ++
    if (RepFrac == 1) then
        RepFrac = Xscale
        if (RdBit = 7) then
            RdBit = 0
            FifoRdPnt ++
            FifoContents --
        else
            RdBit++
    else
        RepFrac--
    if(BitToTx == 1) then {
        RepFrac = XFracScale
        RdBit = 0
        FifoRdPnt ++
        FifoContents--
        BitToTx = HCUDotCount
        }
```

)

What is not detailed above is the fact that, since this is a circular buffer, both the fifo read and write-pointers wrap-around to zero after they reach two. Also not detailed is the fact that if there is a change of both the read and write-pointer in the same cycle, the fifo contents counter remains unchanged.

28   Halftoner Compositor Unit (HCU)

28.1   OVERVIEW

The Halftoner Compositor Unit (HCU) produces dots for each nozzle in the destination printhead taking account of the page dimensions (including margins). The spot data and tag data are received in bi-level form while the pixel contone data received from the CFU must be dithered to a bi-level representation. The resultant 6 bi-level planes for each dot position on the page are then remapped to 6 output planes and output dot at a time (6 bits) to the next stage in the printing pipeline, namely the dead nozzle compensator (DNC).

28.2   DATA FLOW

Figure 194 shows a simple dot data flow high level block diagram of the HCU. The HCU reads contone data from the CFU, bi-level spot data from the SFU, and bi-level tag data from the TFU. Dither matrices are read from the DRAM via the DIU. The calculated output dot (6 bits) is read by the DNC
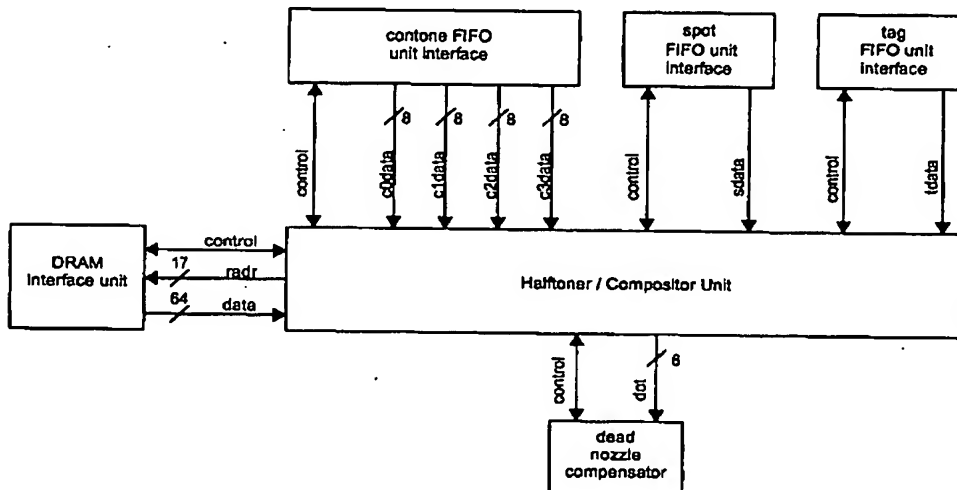
Figure 194. High level block diagram showing the HCU and its external interfaces

The HCU is given the page dimensions (including margins), and is only started once for the page. It does not need to be programmed in between bands or restarted for each band. The HCU will stall appropriately if its input buffers are starved. At the end of the page the HCU will continue to produce 0 for all dots as long as data is requested by the units further down the pipeline (this allows later units to conveniently flush pipelined data).

The HCU performs a linear processing of dots calculating the 6-bit output of a dot in each cycle. The mapping of 6 calculated bits to 6 output bits for each dot allows for such example mappings as compositing of the spot0 layer over the appropriate contone layer (typically black), the merging of CMY into K (if K is present in the printhead), the splitting of K into CMY dots if there is no K in the printhead, and the generation of a fixative output bitstream.

28.3   DRAM STORAGE REQUIREMENTS

SoPEC allows for a number of different dither matrix configurations up to 256 bytes wide. The dither matrix is stored in DRAM. Using either a single or double-buffer scheme a line of the dither matrix must be read in by the HCU over a SoPEC line time. SoPEC must produce 13824 dots per line for A4/Letter printing which takes 13824 cycles.

The following give the storage and bandwidths requirements for some of the possible configurations of the dither matrix.

- 4 Kbyte DRAM storage required for one 64x64 (preferred) byte dither matrix
- 6.25 Kbyte DRAM storage required for one 80x80 byte dither matrix
- 16 Kbyte DRAM storage required for four 64x64 byte dither matrices
- 64 Kbyte DRAM storage required for one 256x256 byte dither matrix

Note that regardless of the width of the dither matrix, 256 bytes are always read from DRAM for each line.

28.4   IMPLEMENTATION

A block diagram of the HCU is given in Figure 195.

Figure 195. Block diagram of the HCU

28.4.1   Definition of I/O

Table 142. HCU port list and description

Port name	Pins	I/O	Description
Clocks and reset			
pclk	1	In	System clock.
prst_n	1	In	System reset, synchronous active low.
PCU Interface			
pcu_hcu_sel	1	In	Block select from the PCU. When *pcu_hcu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
hcu_pcu_rdy	1	Out	Ready signal to the PCU. When *hcu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *hcu_pcu_data* is valid.
hcu_pcu_data[31:0]	32	Out	Read data bus to the PCU.
DIU Interface			
hcu_diu_rreq	1	Out	HCU read request, active high. A read request must be accompanied by a valid read address.
diu_hcu_rack	1	In	Acknowledge from DIU, active high. Indicates that a read request has been accepted and the new read address can be placed on the address bus, *hcu_diu_radr*.
hcu_diu_radr[21:5]	17	Out	HCU read address. 17 bits wide (256-bit aligned word).
diu_hcu_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DIU.
CFU Interface			
cfu_hcu_avail	1	In	Indicates valid data present on cfu_hcu_c[3-0]data lines.
cfu_hcu_c0data[7:0]	8	In	Pixel of data in contone plane 0.
cfu_hcu_c1data[7:0]	8	In	Pixel of data in contone plane 1.
cfu_hcu_c2data[7:0]	8	In	Pixel of data in contone plane 2.
cfu_hcu_c3data[7:0]	8	In	Pixel of data in contone plane 3.
hcu_cfu_advdot	1	Out	Informs the CFU that the HCU has captured the pixel data on *cfu_hcu_c[3-0]data* lines and the CFU can now place the next pixel on the data lines.
SFU Interface			
sfu_hcu_avail	1	In	Indicates valid data present on *sfu_hcu_sdata*.
sfu_hcu_sdata	1	In	Bi-level dot data.
hcu_sfu_advdot	1	Out	Informs the SFU that the HCU has captured the dot data on *sfu_hcu_sdata* and the SFU can now place the next dot on the data line.
TFU Interface			
tfu_hcu_avail	1	In	Indicates valid data present on *tfu_hcu_tdata*.
tfu_hcu_tdata	1	In	Tag dot data.

Table 142. HCU port list and description

Port name	Pins	I/O	Description
hcu_tfu_advdot	1	Out	Informs the TFU that the HCU has captured the dot data on *tfu_hcu_tdata* and the TFU can now place the next dot on the data line.
DNC interface			
dnc_hcu_ready	1	In	Indicates that DNC is ready to accept data from the HCU.
hcu_dnc_avail	1	Out	Indicates valid data present on *hcu_dnc_data*.
hcu_dnc_data[5:0]	6	Out	Output bi-level dot data in 6 ink planes.

28.4.2 Configuration Registers

The configuration registers in the HCU are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for the description of the protocol and timing diagrams for reading and writing registers in the HCU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the HCU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *hcu_pcu_data*. The configuration registers of the HCU are listed in Table 143.

Table 143. HCU Registers

Address (HCU base)	Register Name	#bits	Value on Reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the HCU.
0x04	Go	1	0x0	Writing 1 to this register starts the HCU. Writing 0 to this register halts the HCU. When *Go* is asserted all counters, flags etc. are cleared or given their initial value, but configuration registers keep their values. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. The HCU should be started *after* the CFU, SFU, TFU, and DNC. This register can be read to determine if the HCU is running (1 = running, 0 = stopped).
Setup registers (constant for during processing)				
0x10	AvailMask	4	0x0	Mask used to determine which of the dotgen units etc. are to be checked before a dot is generated by the HCU within the specified margins for the specified color plane. If the specified dotgen unit is stalled, then the HCU will also stall. See Table 144 for bit allocation and definition.
0x14	TMMask	4	0x0	Same as AvailMask, but used in the top margin area before the appropriate target page is reached.
0x18	PageMarginY	32	0x0000_0000	The first line considered to be off the page.
0x1C	MaxDot	16	0x0000	This is the maximum dot number - 1 present across a page. For example if a page contains 13824 dots, then *MaxDot* will be 13823.

Table 143. HCU Registers

Address (HCU-base)	Register Name	#bits	Value on Reset	Description
0x20	TopMargin	32	0x0000_0000	The first line on a page to be considered within the target page for contone and spot data. (0 = first printed line of page)
0x24	BottomMargin	32	0x0000_0000	The first line in the target bottom margin for contone and spot data (i.e. first line after target page).
0x28	LeftMargin	16	0x0000	The first dot on a line within the target page for contone and spot data.
0x2C	RightMargin	16	0xFFFF	The first dot on a line within the target right margin for contone and spot data.
0x30	TagTopMargin	32	0x0000_0000	The first line on a page to be considered within the target page for tag data. (0 = first printed line of page)
0x34	TagBottomMargin	32	0x0000_0000	The first line in the target bottom margin for tag data (i.e. first line after target page).
0x38	TagLeftMargin	16	0x0000	The first dot on a line within the target page for tag data.
0x3C	TagRightMargin	16	0xFFFF	The first dot on a line within the target right margin for tag data.
0x40	DMReadEnable	1	0x0	1 if a dither matrix is specified 0 if a dither matrix is not specified.
0x44	StartDMAdr	17	0x0_0000	Points to the first 256-bit word of the first line of the dither matrix in DRAM.
0x48	EndDMAdr	17	0x0_0000	Points to the last 256-bit word of the last line of the dither matrix in DRAM.
0x4C	LineIncrement	5	0x2	The number of 256-bit words in DRAM from the start of one line of the dither matrix and the start of the next line, i.e. the value by which the DRAM address is incremented at the start of a line so that it points to the start of the next line of the dither matrix.
0x50	DMInitIndexC0	8	0x00	Initial index within 256-byte dither matrix line buffer for contone plane 0. If using double-buffer scheme, only the 7 lsbs are used.
0x54	DMLwrIndexC0	8	0x00	Lower index within 256-byte dither matrix line buffer for contone plane 0. If using double-buffer scheme, only the 7 lsbs are used.
0x58	DMUprIndexC0	8	0x3F	Upper index within 256-byte dither matrix line buffer for contone plane 0. After reading the data at this location the index wraps to *DMLwrIndexC0*. If using double-buffer scheme, only the 7 lsbs are used.
0x5C	DMInitIndexC1	8	0x00	Initial index within 256-byte dither matrix line buffer for contone plane 1. If using double-buffer scheme, only the 7 lsbs are used.
0x60	DMLwrIndexC1	8	0x00	Lower index within 256-byte dither matrix line buffer for contone plane 1. If using double-buffer scheme, only the 7 lsbs are used.
0x64	DMUprIndexC1	8	0x3F	Upper index within 256-byte dither matrix line buffer for contone plane 1. After reading the data at this location the index wraps to *DMLwrIndexC1*. If using double-buffer scheme, only the 7 lsbs are used.

Table 143. HCU Registers

Address (HCU base+)	Register Name	#bits	Value on Reset	Description
0x68	DMinitIndexC2	8	0x00	Initial index within 256-byte dither matrix line buffer for contone plane 2. If using double-buffer scheme, only the 7 lsbs are used.
0x6C	DMLwrIndexC2	8	0x00	Lower index within 256-byte dither matrix line buffer for contone plane 2. If using double-buffer scheme, only the 7 lsbs are used.
0x70	DMUprIndexC2	8	0x3F	Upper index within 256-byte dither matrix line buffer for contone plane 2. After reading the data at this location the index wraps to *DMLwrIndexC2*. If using double-buffer scheme, only the 7 lsbs are used.
0x74	DMinitIndexC3	8	0x00	Initial index within 256-byte dither matrix line buffer for contone plane 3. If using double-buffer scheme, only the 7 lsbs are used.
0x78	DMLwrIndexC3	8	0x00	Lower index within 256-byte dither matrix line buffer for contone plane 3. If using double-buffer scheme, only the 7 lsbs are used.
0x7C	DMUprIndexC3	8	0x3F	Upper index within 256-byte dither matrix line buffer for contone plane 3. After reading the data at this location the index wraps to *DMLwrIndexC3*. If using double-buffer scheme, only the 7 lsbs are used.
0x80	DoubleLineBuf	1	0x1	Selects the dither line buffer mode to be single or double buffer.
0x84 to 0x98	IOMappingLo	6x32	0x0000_0000	The dot reorg mapping for output inks 0 to 5. For each ink's 64-bit IOMapping value, IOMappingLo represents the low order 32 bits.
0x9C to 0xB0	IOMappingHi	6x32	0x0000_0000	The dot reorg mapping for output inks 0 to 5. For each ink's 64-bit IOMapping value, IOMappingHi represents the high order 32 bits.
0xB4 to 0xC0	cpConstant	4x8	0x00	The constant contone value to output for contone plane N when printing in the margin areas of the page. This value will typically be 0.
0xC4	sConstant	1	0x0	The constant bi-level value to output for spot when printing in the margin areas of the page. This value will typically be 0.
0xC8	tConstant	1	0x0	The constant bi-level value to output for tag data when printing in the margin areas of the page. This value will typically be 0.
0xCC	DitherConstant	8	0xFF	The constant value to use for dither matrix when the dither matrix is not available, i.e. when the signal *dm_avail* is 0. This value will typically be 0xFF so that *cpConstant* can easily be 0x00 or 0xFF without requiring a dither matrix (*DitherConstant* is primarily used for threshold dithering in the margin areas).
Debug registers (read only)				

Table 143. HCU Registers

Address (HCU base +)	Register Name	#bits	Value on Reset	Description
0xD0	HcuPortsDebug	14	N/A	Bit 13 = *tfu_hcu_avail* Bit 12 = *hcu_tfu_advdot* Bit 11 = *sfu_hcu_avail* Bit 10 = *hcu_sfu_advdot* Bit 9 = *cfu_hcu_avail* Bit 8 = *hcu_cfu_advdot* Bit 7 = *dnc_hcu_ready* Bit 6 = *hcu_dnc_avail* Bits 5-0 = *hcu_dnc_data*
0xD4	HcuDotgenDebug	15	N/A	Bit 14 = *after_top_margin* Bit 13 = *in_tag_target_page* Bit 12 = *in_target_page* Bit 11 = *tp_avail* Bit 10 = *s_avail* Bit 9 = *cp_avail* Bit 8 = *dm_avail* Bit 7 = *advdot* Bits 5-0 = [*tp,s,cp3,cp2,cp1,cp0*] (i.e. 6 bit input to dot reorg units)
0xD8	HcuDitherDebug1	17	N/A	Bit 9 = *advdot* Bit 8 = *dm_avail* Bit 15-8 = *cp1_dither_val* Bits 7-0 = *cp0_dither_val*
0xDC	HcuDitherDebug2	17	N/A	Bit 9 = *advdot* Bit 8 = *dm_avail* Bit 15-8 = *cp3_dither_val* Bits 7-0 = *cp2_dither_vall*

28.4.3 Control unit

The control unit is responsible for controlling the overall flow of the HCU. It is responsible for determining whether or not a dot will be generated in a given cycle, and what dot will actually be generated - including whether or not the dot is in a margin area, and what dither cell values should be used at the specific dot location. A block diagram of the control unit is shown in Figure 196.

The inputs to the control unit are a number of avail flags specifying whether or not a given dotgen unit is capable of supplying 'real' data in this cycle. The term 'real' refers to data generated from external sources, such as contone line buffers, bi-level line buffers, and tag plane buffers. Each dotgen unit informs the control unit whether or not a dot can be generated this cycle from real data. It must also check that the DNC is ready to receive data.

The contone/spot margin unit is responsible for determining whether the current dot coordinate is within the target contone/spot margins, and the tag margin unit is responsible for determining whether the current dot coordinate is within the target tag margins.

The dither matrix table interface provides the interface to DRAM for the generation of dither cell values that are used in the halftoning process in the contone dotgen unit.
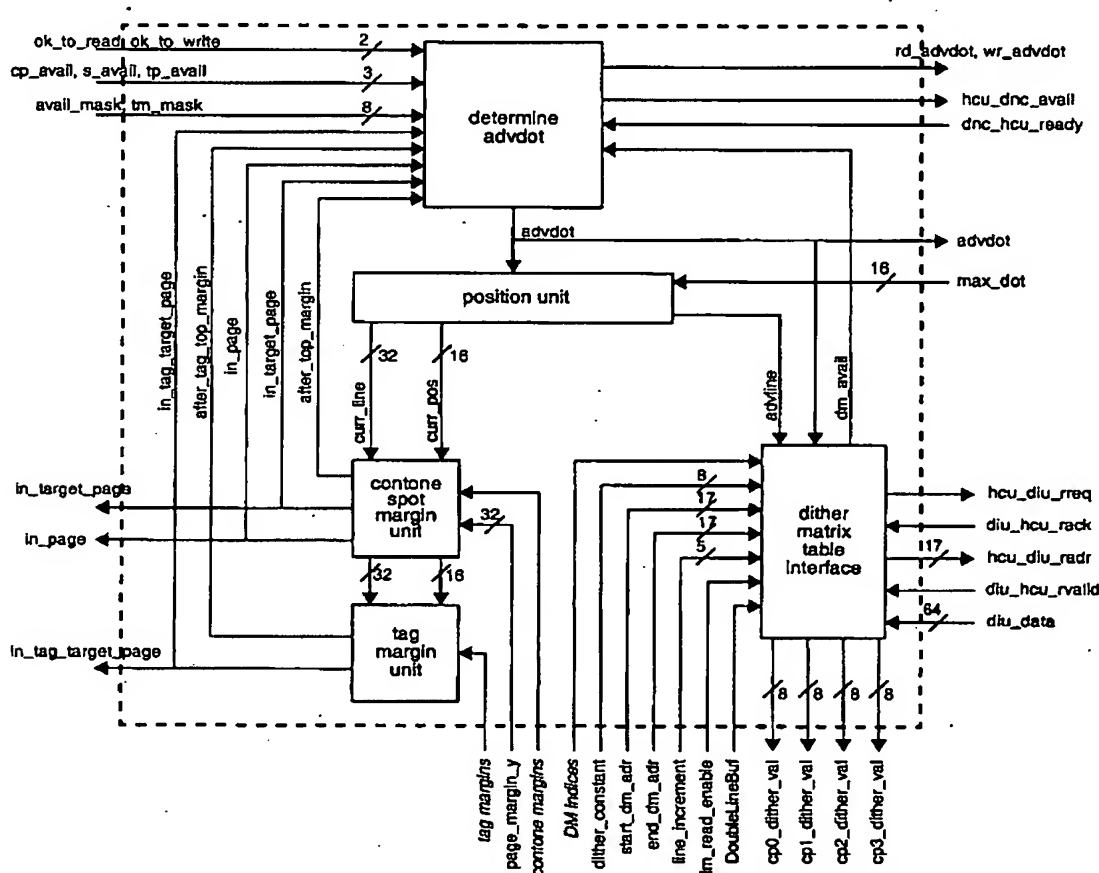
Figure 196. Block diagram of the control unit

28.4.3.1 Determine AdvDot

The HCU does not always require contone planes, bi-level or tag planes in order to produce a page. For example, a given page may not have a bi-level layer, or a tag layer. In addition, the contone and bi-level parts of a page are only required within the contone and bi-level page margins, and the tag part of a page is only required within the tag page margins. Thus output dots can be generated without contone, bi-level or tag data before the respective top margins of a page has been reached, and 0s are generated for all color planes after the end of the page has been reached (to allow later stages of the printing pipeline to flush).

Consequently the HCU has an *AvailMask* register that determines which of the various input avail flags should be taken notice of during the production of a page from the first line of the target page, and a *TMMask* register that has the same behaviour, but is used in the lines before the target page has been reached (i.e. inside the target top margin area). Each bit in the *AvailMask* refers to a particular avail bit: if the bit in the *AvailMask* register is set, then the corresponding *avail* bit must be 1 for the HCU to advance a dot. The bit to avail correspondence is shown in Table 144. Care should be taken with *TMMask* - if the particular data is not available after the top margin has been reached, then the HCU will stall. Note that the *avail* bits for contone and spot colors are ANDed with *in_target_page* after the target page area has been reached to allow dot production in the contone/spot margin areas without needing any data in the CFU and

SFU. The *avail* bit for tag color is ANDed with *in_tag_target_page* after the target tag page area has been reached to allow dot production in the tag margin areas without needing any data in the TFU.

Table 144. Correspondence between bit in AvailMask and avail flag

bit in AvailMask	avail flag	description
0	dm_avail	dither matrix data available
1	cp_avail	contone pixels available
2	s_avail	spot color available
3	tp_avail	tag plane available

Each of the input *avail* bits is processed with its appropriate mask bit and the *after_top_margin* flag. The output bits are ANDed together along with *Go* and *ok_to_write* (which specifies whether the output buffer is ready to receive a dot in this cycle) to form the output bit *advdot*. We also generate *wr_advdot*. In this way, if the output buffer is full or any of the specified avail flags is clear, the HCU will stall. When the end of the page is reached, *in_page* will be deasserted and the HCU will continue to produce 0 for all dots as long as the DNC requests data. A block diagram of the determine advdot unit is shown in Figure 197.

The *ok_to_read* signal from the output buffer indicates that the HCU has a dot available for the DNC to read (indicated to the DNC by the assertion of *hcu_dnc_avail*). If the DNC is ready to receive the dot (*dnc_hcu_ready* is 1) then the dot is read from the output buffer by asserting *rd_advdot*.
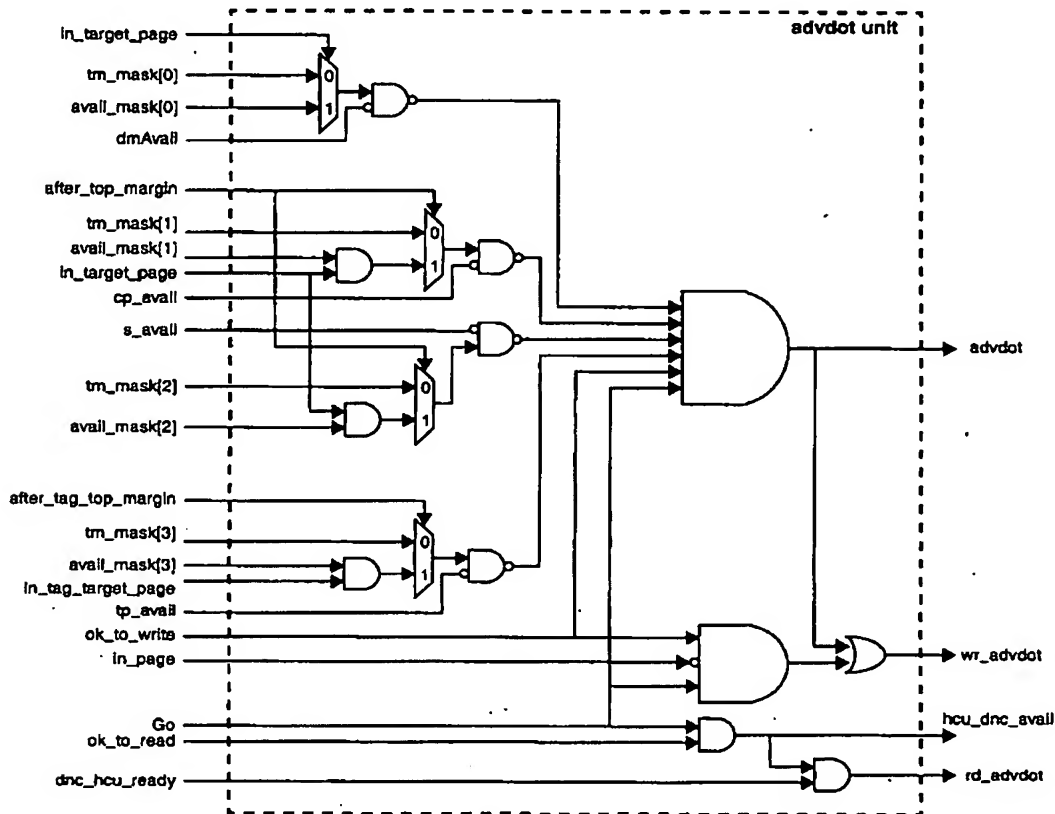
Figure 197. Block diagram of determine advdot unit

28.4.3.2 Position unit

The position unit is responsible for outputting the position of the current dot (*curr_pos*, *curr_line*) and whether or not this dot is the last dot of a line (*advline*). Both *curr_pos* and *curr_line* are set to 0 at reset or when *Go* transitions from 0 to 1. The position unit relies on the *advdot* input signal to advance through the dots on a page. Whenever an *advdot* pulse is received, *curr_pos* gets incremented. If *curr_pos* equals *max_dot* then an *advline* pulse is generated as this is the last dot in a line, *curr_line* gets incremented, and the *curr_pos* is reset to 0 to start counting the dots for the next line.

28.4.3.3 Margin unit

The responsibility of the margin unit is to determine whether the specific dot coordinate is within the page at all, within the target page or in a margin area (see Figure 198). This unit is instantiated for both the contone/spot margin unit and the tag margin unit.
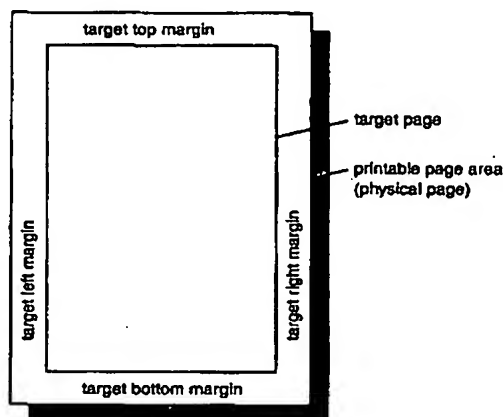
Figure 198. Page structure

The margin unit takes the current dot and line position, and returns three flags.

- the first, *in_page* is 1 if the current dot is within the page, and 0 if it is outside the page.
- the second flag, *in_target_page*, is 1 if the dot coordinate is within the target page area of the page, and 0 if it is within the target top/left/bottom/right margins.
- the third flag, after_top_margin, is 1 if the current dot is below the target top margin, and 0 if it is within the target top margin.

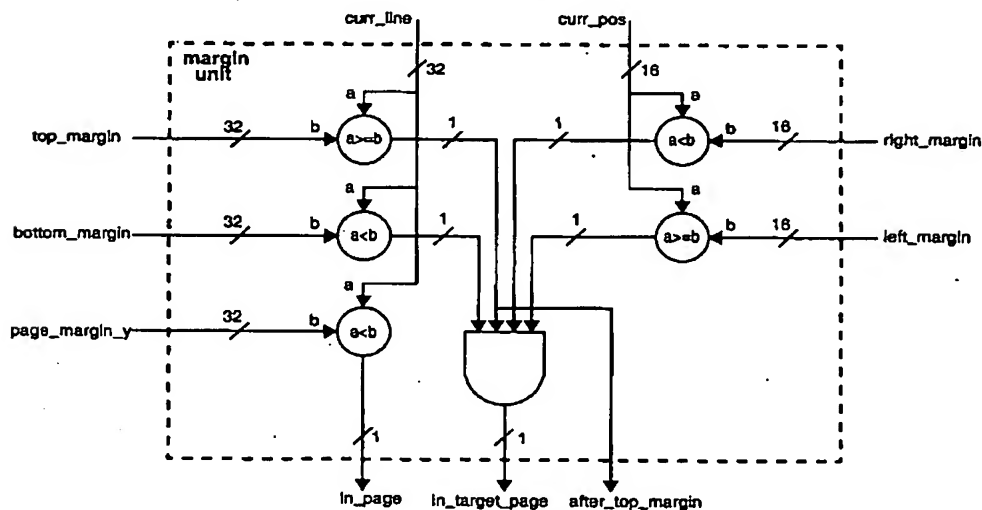A block diagram of the margin unit is shown in Figure 199.

Figure 199. Block diagram of margin unit

28.4.3.4 Dither matrix table interface

The dither matrix table interface provides the interface to DRAM for the generation of dither cell values that are used in the halftoning process in the contone dotgen unit. The control flag *dm_read_enable* enables the reading of the dither matrix table line structure from DRAM. If *dm_read_enable* is 0, the dither matrix is not specified in DRAM and no DRAM accesses are attempted. The dither matrix table interface has an output flag *dm_avail* which specifies if the current line of the specified matrix is available. The HCU can be directed to stall when *dm_avail* is 0 by setting the appropriate bit in the HCU's *Avail-Mask* or *TMMask* registers. When *dm_avail* is 0 the value in the *DitherConstant* register is used as the dither cell values that are output to the contone dotgen unit.

The dither matrix table interface consists of a state machine that interfaces to the DRAM interface, a dither matrix buffer that provides dither matrix values, and a unit to generate the addresses for reading the buffer. Figure 200 shows a block diagram of the dither matrix table interface.
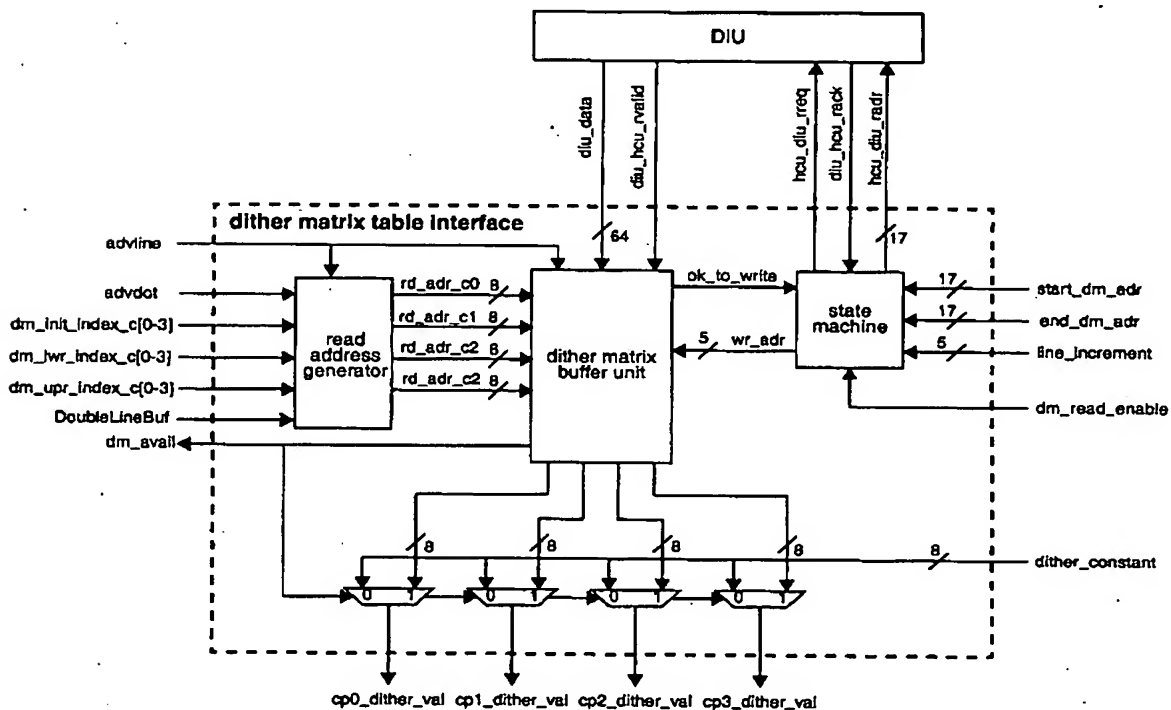

Figure 200. Block diagram of dither matrix table interface

28.4.3.4.1 Dither matrix buffer

The state machine loads dither matrix table data a line at a time from DRAM and stores it in a buffer. A single line of the dither matrix is either 256 or 128 8-bit entries, depending on the programmable bit DoubleLineBuf. If this bit is enabled, a double-buffer mechanism is employed such that while one buffer is read from for the current line's dither matrix data (8 bits representing a single dither matrix entry), the other buffer is being written to with the next line's dither matrix data (64-bits at a time). Alternatively, the

single buffer scheme can be used, where the data must be loaded at the end of the line, thus incurring a delay.

The single/double buffer is implemented using a 256 byte 3-port register array, two reads, one write port, with the reads clocked at double the system clock rate (320MHz) allowing 4 reads per clock cycle.

The dither matrix buffer unit also provides the mechanism for keeping track of the current read and write buffers, and providing the mechanism such that a buffer cannot be read from until it has been written to. In this case, each buffer is a line of the dither matrix, i.e. 256 or 128 bytes.

A bit is kept for the status of each dither matrix line buffer: *buff_avail[0]* and *buff_avail[1]*. It also keeps a single bit (*rd_buff*) for the current buffer that reads are to occur from, and a single bit (*wr_buff*) for the current buffer that writes are to occur to. The output value *dm_avail* equals *buff_avail[rd_buff]*. The output value *ok_to_write* equals *buff_avail[wr_buff]*. Note that when using a single line buffer, *buff_avail[1]* is not used.

The read addresses are byte aligned. A single dither matrix entry is represented by 8 bits and an entry is read for each of the four contone planes in parallel. When a *advline* pulse is received, *buff_avail[rd_buff]* is cleared, and *rd_buff* is inverted (if using a double line buffer).

Data is written, 64 bits at a time to the current write buffer when *diu_hcu_rvalid* is asserted. When *WrAdr* is 0x1F and *diu_hcu_rvalid* is 1, *buff_avail[wr_buff]* is set, and *wr_buff* is inverted (if using a double line buffer). This indicates that a line of dither matrix has been written to the current write buffer and it is now available to be read.

28.4.3.4.2 Read address generator

For each contone plane there is a initial, lower and upper index to be used when reading dither cell values from the dither matrix double buffer. The read address for each plane is used to select a byte from the current 256-byte read buffer. When *Go* gets set (0 to 1 transition), or at the end of a line, the read addresses are set to their corresponding initial index. Otherwise, the read address generator relies on *advdot* to advance the addresses within the inclusive range specified the lower and upper indices, represented by the following pseudocode:

```
if (advdot == 1) then
    if (advline == 1) then
        rd_adr = dm_init_index
    elsif (rd_adr == dm_upr_index) then
        rd_adr = dm_lwr_index
    else
        rd_adr ++
else
    rd_adr = rd_adr
```

28.4.3.4.3 State machine

The dither matrix is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle).The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 208. Read accesses to DRAM are implemented by means of the state machine described in Figure 201.

All counters and flags should be cleared after reset or when *Go* transitions from 0 to 1. While the *Go* bit is 1, the state machine relies on the *dm_read_enable* bit to tell it whether to attempt to read dither matrix data from DRAM. When *dm_read_enable* is clear, the state machine does nothing and remains in the idle state. When *dm_read_enable* is set, the state machine continues to load dither matrix data, 256-bits at a time (received over 4 clock cycles, 64 bits per cycle), while there is space available in the dither matrix buffer.

The read address and *line_start_adr* are initially set to *start_dm_adr.* The read address gets incremented after each read access. It takes 4 or 8 read accesses to load a line of dither matrix into the dither matrix buffer, depending on whether we're using a single or double buffer. A count is kept of the accesses to DRAM. When a read access completes and *access_count* equals 3 or 7, a line of dither matrix has just been loaded from and the read address is updated to *line_start_adr* plus *line_increment* so it points to the start of the next line of dither matrix. *(line_start_adr* is also updated to this value). If the read address equals *end_dm_adr* then the next read address will be *start_dm_adr*, thus the read address wraps to point to the start of the area in DRAM where the dither matrix is stored.

The write address for the dither matrix buffer is implemented by means of a modulo-32 counter that is initially set to 0 and incremented when *diu_hcu_rvalid* is asserted.
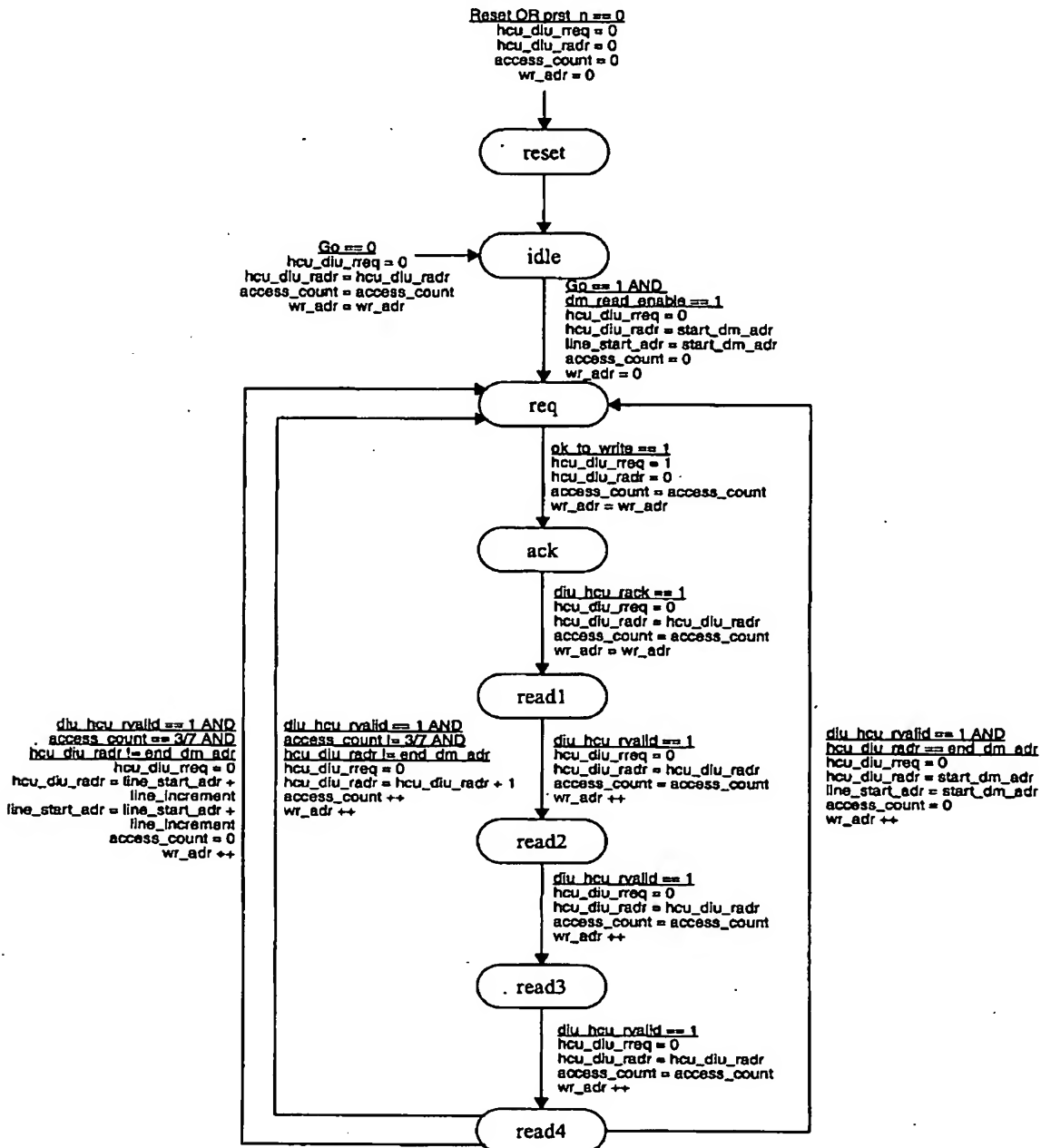
Figure 201. State machine to read dither matrix table

28.4.4   Contone dotgen unit

The contone dotgen unit is responsible for producing a dot in up to 4 color planes per cycle. The contone dotgen unit also produces a *cp_avail* flag which specifies whether or not contone pixels are currently available, and the output *hcu_cfu_advdot* to request the CFU to provide the next contone pixel in up to 4 color planes.

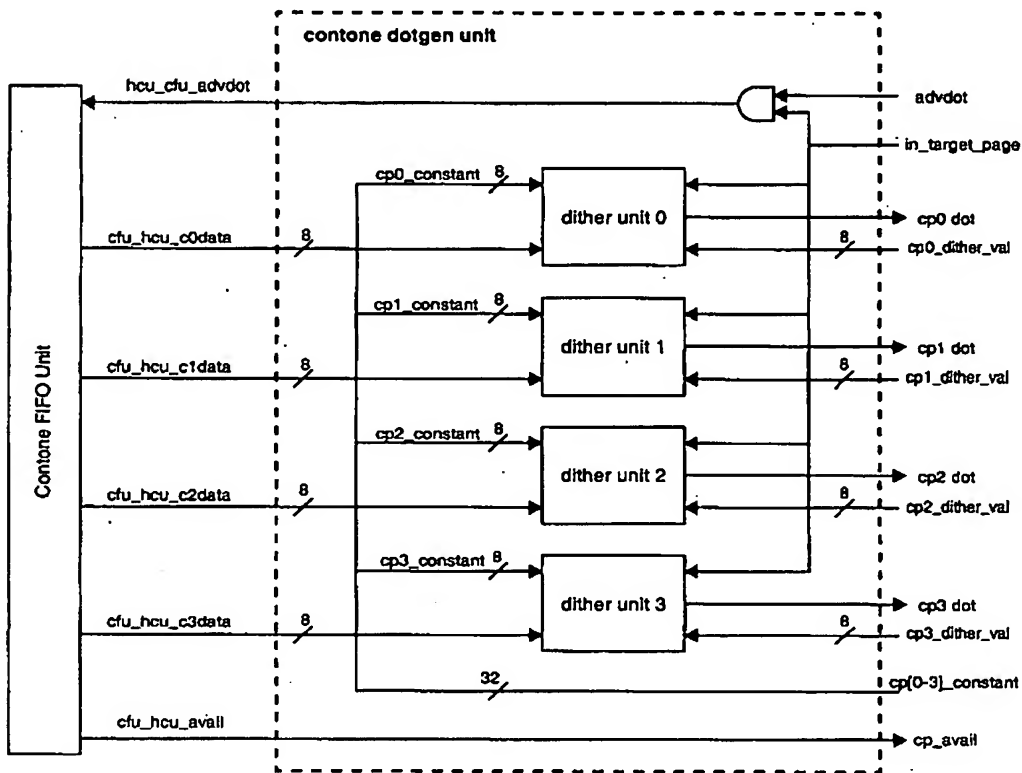The block diagram for the contone dotgen unit is shown in Figure 202.

Figure 202. Contone dotgen unit

A dither unit provides the functionality for dithering a single contone plane. The contone image is only defined within the contone/spot margin area. As a result, if the input flag *in_target_page* is 0, then a constant contone pixel value is used for the pixel instead of the contone plane.

The resultant contone pixel is then halftoned. The dither value to be used in the halftoning process is provided by the control data unit. The halftoning process involves a comparison between a pixel value and its corresponding dither value. If the 8-bit contone value is *greater than or equal* to the 8-bit dither matrix value a 1 is output. If not, then a 0 is output. This means each entry in the dither matrix is in the range 1-255 (0 is not used).

28.4.5   Spot dotgen unit

The spot dotgen unit is responsible for producing a dot of bi-level data per cycle. It deals with bi-level data (and therefore does not need to halftone) that comes from the LBD via the SFU. Like the contone layer, the bi-level spot layer is only defined within the contone/spot margin area. As a result, if input flag *in_target_page* is 0, then a constant dot value (typically this would be 0) is used for the output dot.

The spot dotgen unit also produces a *s_avail* flag which specifies whether or not spot dots are currently available for this spot plane, and the output *hcu_sfu_advdot* to request the SFU to provide the next bi-level data value. The spot dotgen unit can be represented by the following pseudocode:

```
s_avail = sfu_hcu_avail

if (in_target_page == 1 AND advdot == 1) then
    hcu_sfu_advdot = 1
else
    hcu_sfu_advdot = 0

if (in_target_page == 1) then
    sp = sfu_hcu_sdata
else
    sp = sp_constant
```

28.4.6   Tag dotgen unit

This unit is very similar to the spot dotgen unit (see Section 28.4.5) in that it deals with bi-level data, in this case from the TE via the TFU. The tag layer is only defined within the tag margin area. As a result, if input flag *in_tag_target_page* is 0, then a constant dot value, *tp_constant* (typically this would be 0), is used for the output dot. The tagplane dotgen unit also produces a *tp_avail* flag which specifies whether or not tag dots are currently available for the tagplane, and the output *hcu_tfu_advdot* to request the TFU to provide the next bi-level data value.

28.4.7   Dot reorg unit

The dot reorg unit provides a means of mapping the bi-level dithered data, the spot0 color, and the tag data to output inks in the actual printhead. Each dot reorg unit takes a set of 6 1-bit inputs and produces a single bit output that represents the output dot for that color plane.

The output bit is a logical combination of any or all of the input bits. This allows the spot color to be placed in any output color plane (including infrared for testing purposes), black to be merged into cyan, magenta and yellow (in the case of no black ink in the Memjet printhead), and tag dot data to be placed in a visible plane. An output for fixative can readily be generated by simply combining desired input bits.

The dot reorg unit contains a 64-bit lookup to allow complete freedom with regards to mapping. Since all possible combinations of input bits are accounted for in the 64 bit lookup, a given dot reorg unit can take the mapping of other reorg units into account. For example, a black plane reorg unit may produce a 1 only if the contone plane 3 or spot color inputs are set (this effectively composites black bi-level over the contone). A fixative reorg unit may generate a 1 if any 2 of the output color planes is set (taking into account the mappings produced by the other reorg units).

If dead nozzle replacement is to be used (see section 29.4.2 on page 448), the dot reorg can be programmed to direct the dots of the specified color into the main plane, and 0 into the other. If a nozzle is then marked as dead in the DNC, swapping the bits between the planes will result in 0 in the dead nozzle, and the required data in the other plane.

If dead nozzle replacement is to be used, and there are no tags, the TE can be programmed with the position of dead nozzles and the resultant pattern used to direct dots into the specified nozzle row. If only fixed

background TFS is to be used, a limited number of nozzles can be replaced. If variable tag data is to be used to specify dead nozzles, then large numbers of dead nozzles can be readily compensated for.

The dot reorg unit can be used to average out the nozzle usage when two rows of nozzles share the same ink and tag encoding is not being used. The TE can be programmed to produce a regular pattern (e.g. 0101 on one line, and 1010 on the next) and this pattern can be used as a directive as to direct dots into the specified nozzle row.

Each reorg unit contains a 64-bit *IOMapping* value programmable as two 32-bit HCU registers, and a set of selection logic based on the 6-bit dot input ($2^6 = 64$ bits), as shown in Figure 203.
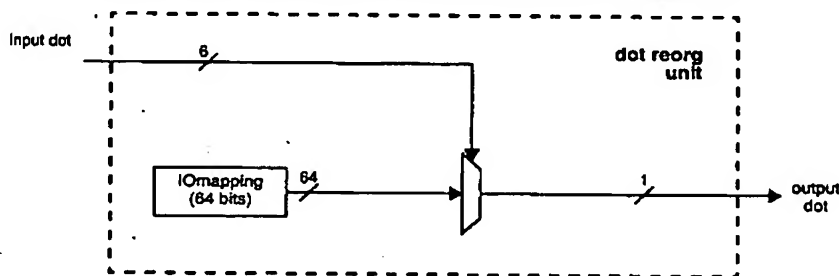
Figure 203. Block diagram of dot reorg unit

The mapping of input bits to each of the 6 selection bits is as defined in Table 145.

Table 145. Mapping of input bits to 6 selection bits

address bit of lookup	dot	likely interpretation
0	bi-level dot from contone layer 0	cyan
1	bi-level dot from contone layer 1	magenta
2	bi-level dot from contone layer 2	yellow
3	bi-level dot from contone layer 3	black
4	bi-level spot0 dot	black
5	bi-level tag dot	infra-red

29 Dead Nozzle Compensator (DNC)

29.1 OVERVIEW

The Dead Nozzle Compensator (DNC) is responsible for adjusting Memjet dot data to take account of non-functioning nozzles in the Memjet printhead. Input dot data is supplied from the HCU, and the corrected dot data is passed out to the DWU. The high level data path is shown by the block diagram in Figure 204.
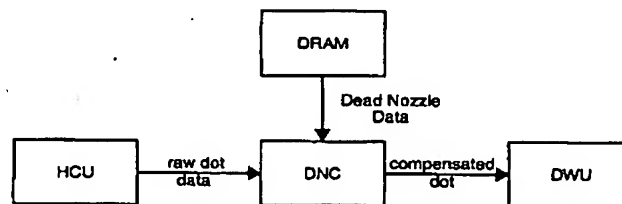

Figure 204. High level block diagram of DNC

The DNC compensates for a dead nozzles by performing the following operations:
*   Dead nozzle removal, i.e. turn the nozzle off
*   Ink replacement by direct substitution i.e. K -> K
*   Ink replacement by indirect substitution i.e. K -> CMY
*   Error diffusion to adjacent nozzles
*   Fixative corrections

The DNC is required to efficiently support up to 5% dead nozzles, under the expected DRAM bandwidth allocation, with no restriction on where dead nozzles are located and handle any fixative correction due to nozzle compensations. Performance must degrade gracefully after 5% dead nozzles.

29.2 DEAD NOZZLE IDENTIFICATION

Dead nozzles are identified by means of a position value and a mask value. Position information is represented by a 10-bit delta encoded format, where the 10-bit value defines the number of dots between dead nozzle columns[1]. With the delta information it also reads the 6-bit dead nozzle mask (*dn_mask*) for the defined dead nozzle position. Each bit in the *dn_mask* corresponds to an ink plane. A set bit indicates that the nozzle for the corresponding ink plane is dead. The dead nozzle table format is shown in Figure 205. The DNC reads dead nozzle information from DRAM in single 256-bit accesses. A 10-bit delta encoding scheme is chosen so that each table entry is 16 bits wide, and 16 entries fit exactly in each 256-bit read. Using 10-bit delta encoding means that the maximum distance between dead nozzle columns is 1023 dots. It is possible that dead nozzles may be spaced further than 1023 dots from each other, so a null dead nozzle identifier is required. A null dead nozzle identifier is defined as a 6-bit *dn_mask* of all zeros. These null dead nozzle identifiers should also be used so that:
*   the dead nozzle table is a multiple of 16 entries (so that it is aligned to the 256-bit DRAM locations)

---

1. for a 10-bit delta value of *d*, if the current column *n* is a dead nozzle column then the next dead nozzle column is given by $n + (d + 1)$.

- the dead nozzle table spans the complete length of the line, i.e. the first entry dead nozzle table should have a delta from the first nozzle column in a line and the last entry in the dead nozzle table should correspond to the last nozzle column in a line.

Note that the DNC deals with the width of a page. This may or may not be the same as the width of the printhead (the PHI may introduce some margining to the page so that its dot output matches the width of the printhead). Care must be taken when programming the dead nozzle table so that dead nozzle positions are correctly specified with respect to the page and printhead.
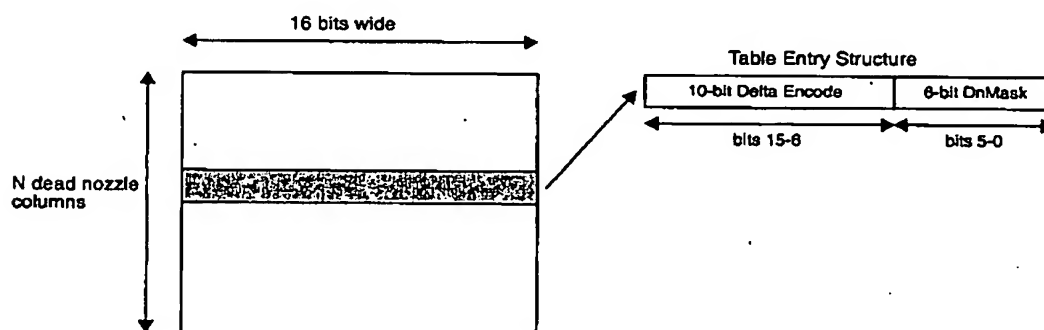

Figure 205. Dead nozzle table format

29.3   DRAM STORAGE AND BANDWIDTH REQUIREMENT

The memory required is largely a factor of the number of dead nozzles present in the printhead (which in turn is a factor of the printhead size). The DNC is required to read a 16-bit entry from the dead nozzle table for every dead nozzle. Table 146 shows the DRAM storage and average[1] bandwidth requirements for the DNC for different percentages of dead nozzles and different page sizes.

Table 146. Dead Nozzle storage and average bandwidth requirements

Page size	% Dead Nozzles	Dead nozzle table	
		Memory (KBytes)	Bandwidth (bits/cycle)
A4[a]	5%	1.4[c]	0.8[d]
	10%	2.7	1.6
	15%	4.1	2.4
A3[b]	5%	1.9	0.8
	10%	3.8	1.6
	15%	5.7	2.4

a. Bi-lithic printhead has 13824 nozzles per color providing full bleed printing for A4/Letter

b. Bi-lithic printhead has 19488 nozzles per color providing full bleed printing for A3

---

1. Average bandwidth assumes an even spread of dead nozzles. Clumps of dead nozzles may cause delays due to insufficient available DRAM bandwidth. These delays will occur every line causing an accumulative delay over a page.

c. 16 bits x 13824 nozzles x 0.05 dead

d. (16 bits read / 20 cycles) = 0.8 bits/cycle

29.4   NOZZLE COMPENSATION

DNC receives 6 bits of dot information every cycle from the HCU, 1 bit per color plane. When the dot position corresponds to a dead nozzle column, the associated 6-bit *dn_mask* indicates which ink plane(s) contains a dead nozzle(s). The DNC first deletes dots destined for the dead nozzle. It then replaces those dead dots, either by placing the data destined for the dead nozzle into an adjacent ink plane (direct substitution) or into a number of ink planes (indirect substitution). After ink replacement, if a dead nozzle is made active again then the DNC performs error diffusion. Finally, following the dead nozzle compensation mechanisms the fixative, if present, may need to be adjusted due to new nozzles being activated, or dead nozzles being removed.

29.4.1   Dead nozzle removal

If a nozzle is defined as dead, then the first action for the DNC is to turn off (zeroing) the dot data destined for that nozzle. This is done by a bit-wise ANDing of the inverse of the *dn_mask* with the dot value.

29.4.2   Ink replacement

Ink replacement is a mechanism where data destined for the dead nozzle is placed into an adjacent ink plane of the same color (direct substitution, i.e. $K \rightarrow K_{alternative}$), or placed into a number of ink planes, the combination of which produces the desired color (indirect substitution, i.e. $K \rightarrow CMY$). Ink replacement is performed by filtering out ink belonging to nozzles that are dead and then adding back in an appropriately calculated pattern. This two step process allows the optional re-inclusion of the ink data into the original dead nozzle position to be subsequently error diffused. In the general case, fixative data destined for a dead nozzle should not be left active intending it to be later diffused.

The ink replacement mechanism has 6 ink replacement patterns, one per ink plane, programmable by the CPU. The dead nozzle mask is ANDed with the dot data to see if there are any planes where the dot is active but the corresponding nozzle is dead. The resultant value forms an enable, on a per ink basis, for the ink replacement process. If replacement is enabled for a particular ink, the values from the corresponding replacement pattern register are ORed into the dot data. The output of the ink replacement process is then filtered so that error diffusion is only allowed for the planes in which error diffusion is enabled. The output of the ink replacement logic is ORed with the resultant dot after dead nozzle removal. See Figure 210 on page 459 for implementation details.

For example if we consider the printhead color configuration C,M,Y,K_1,K_2,IR and the input dot data from the HCU is b101100. Assuming that the K_1 ink plane and IR ink plane for this position are dead so the dead nozzle mask is b000101. The DNC first removes the dead nozzle by zeroing the K_1 plane to produce b101000. Then the dead nozzle mask is ANDed with the dot data to give b000100 which selects the ink replacement pattern for K_1 (in this case the ink replacement pattern for K_1 is configured as b000010, i.e. ink replacement into the K_2 plane). Providing error diffusion for K_2 is enabled, the output from the ink replacement process is b000010. This is ORed with the output of dead nozzle removal to produce the resultant dot b101010. As can be seen the dot data in the defective K_1 nozzle was removed and replaced by a dot in the adjacent K_2 nozzle in the same dot position, i.e. direct substitution.

In the example above the K_1 ink plane could be compensated for by indirect substitution, in which case ink replacement pattern for K_1 would be configured as b111000 (substitution into the CMY color planes), and this is ORed with the output of dead nozzle removal to produce the resultant dot b111000. Here the dot data in the defective K_1 ink plane was removed and placed into the CMY ink planes.

29.4.3   Error diffusion

Based on the programming of the lookup table the dead nozzle may be left active after ink replacement. In such cases the DNC can compensate using error diffusion. Error diffusion is a mechanism where dead nozzle dot data is diffused to adjacent dots.

When a dot is active and its destined nozzle is dead, the DNC will attempt to place the data into an adjacent dot position, if one is inactive. If both dots are inactive then the choice is arbitrary, and is determined by a pseudo random bit generator. If both neighbor dots are already active then the bit cannot be compensated by diffusion.

Since the DNC needs to look at neighboring dots to determine where to place the new bit (if required), the DNC works on a set of 3 dots at a time. For any given set of 3 dots, the first dot received from the HCU is referred to as dot A, and the second as dot B, and the third as dot C. The relationship is shown in Figure 206.
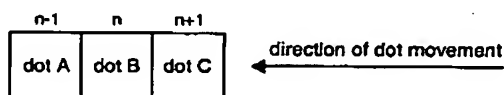

Figure 206. Set of dots operated on for error diffusion

For any given set of dots ABC, only B can be compensated for by error diffusion if B is defined as dead. A 1 in dot B will be diffused into either dot A or dot C if possible. If there is already a 1 in dot A or dot C then a 1 in dot B cannot be diffused into that dot.

The DNC must support adjacent dead nozzles. Thus if dot A is defined as dead and has previously been compensated for by error diffusion, then the dot data from dot B should not be diffused into dot A. Similarly, if dot C is defined as dead, then dot data from dot B should not be diffused into dot C.

Error diffusion should not cross line boundaries. If dot B contains a dead nozzle and is the first dot in a line then dot A represents the last dot from the previous line. In this case an active bit on a dead nozzle of dot B should not be diffused into dot A. Similarly, if dot B contains a dead nozzle and is the last dot in a line then dot C represents the first dot of the next line. In this case an active bit on a dead nozzle of dot B should not be diffused into dot C.

Thus, as a rule, a 1 in dot B cannot be diffused into dot A if
- a 1 is already present in dot A,
- dot A is defined as dead,
- or dot A is the last dot in a line.

Similarly, a 1 in dot B cannot be diffused into dot C if
- a 1 is already present in dot C,
- dot C is defined as dead,
- or dot C is the first dot in a line.

If B is defined to be dead and the dot value for B is 0, then no compensation needs to be done and dots A and C do not need to be changed.

If B is defined to be dead and the dot value for B is 1, then B is changed to 0 and the DNC attempts to place the 1 from B into either A or C:
- If the dot can be placed into both A and C, then the DNC must choose between them. The preference is given by the current output from the random bit generator, 0 for "prefer left" (dot A) or 1 for "prefer right" (dot C).

- If dot can be placed into only one of A and C, then the 1 from B is placed into that position.
- If dot cannot be placed into either one of A or C, then the DNC cannot place the dot in either position.

Table 147 shows the truth table for DNC error diffusion operation when dot B is defined as dead.

Table 147. Error Diffusion Truth Table when dot B is dead

A OR A dead OR A last in line	B	C OR C dead OR C first in line	Rand[a]	A	B	C
0	0	0	X	A input	0	C input
0	0	1	X	A input	0	C input
0	1	0	0	**1**[b]	0	C input
0	1	0	1	A input	0	1
0	1	1	X	1	0	C input
1	0	0	X	A input	0	C input
1	0	1	X	A input	0	C input
1	1	0	X	A input	0	1
1	1	1	X	A input	0	C input

a. Output from random bit generator. Determines direction of error diffusion (0 = left, 1 = right)
b. Bold emphasis is used to show the DNC inserted a 1

The random bit value used to arbitrarily select the direction of diffusion is generated by a 32-bit maximum length random bit generator. The generator generates a new bit for each dot in a line regardless of whether the dot is dead or not. The random bit generator can be initialized with a 32-bit programmable seed value.

29.4.4 Fixative correction

After the dead nozzle compensation methods have been applied to the dot data, the fixative, if present, may need to be adjusted due to new nozzles being activated, or dead nozzles being removed. For each output dot the DNC determines if fixative is required (using the *FixativeRequiredMask* register) for the new compensated dot data word and whether fixative is activated already for that dot. For the DNC to do so it needs to know the color plane that has fixative, this is specified by the *FixativeMask1* configuration register. Table 148 indicates the actions to take based on these calculations.

Table 148. Truth table for fixative correction

Fixative Present	Fixative Required	Action
1	1	Output dot as is.
1	0	Clear fixative plane.
0	1	Attempt to add fixative.
0	0	Output dot as is.

The DNC also allows the specification of another fixative plane, specified by the *FixativeMask2* configuration register, with *FixativeMask1* having the higher priority over *FixativeMask2*. When attempting to add fixative the DNC first tries to add it into the planes defined by *FixativeMask1*. However, if any of these planes is dead then it tries to add fixative by placing it into the planes defined by *FixativeMask2*.

Note that the fixative defined by *FixativeMask1* and *FixativeMask2* could possibly be multi-part fixative, i.e. 2 bits could be set in *FixativeMask1* with the fixative being a combination of both inks.

29.5 IMPLEMENTATION
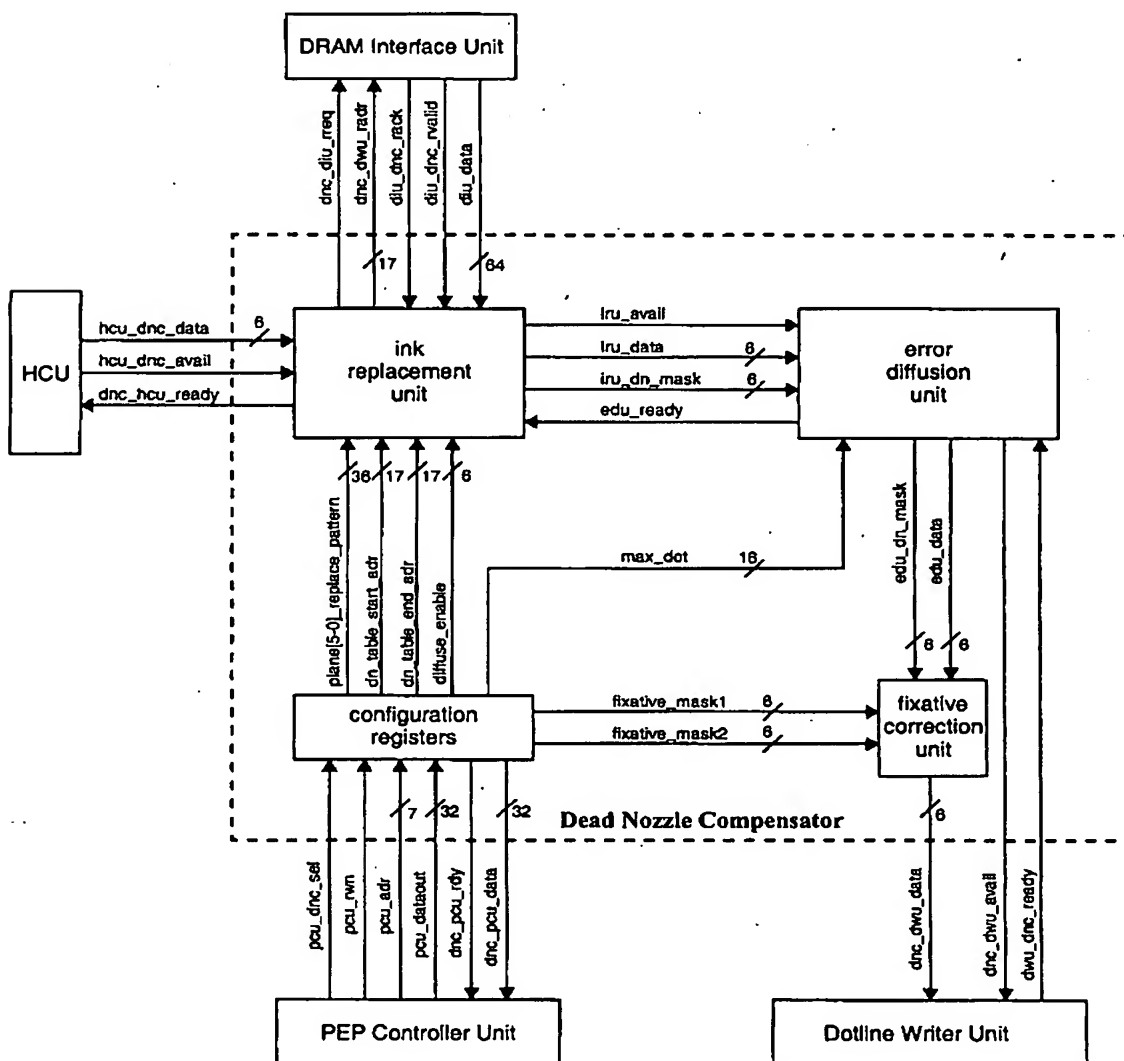
A block diagram of the DNC is shown in Figure 207.

Figure 207. Block diagram of DNC

29.5.1   Definitions of I/O

Table 149. DNC port list and description

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock.
prst_n	1	In	System reset, synchronous active low.
PCU Interface			
pcu_dnc_sel	1	In	Block select from the PCU. When *pcu_dnc_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[6:2]	5	In	PCU address bus. Only 5 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
dnc_pcu_rdy	1	Out	Ready signal to the PCU. When *dnc_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *dnc_pcu_data* is valid.
dnc_pcu_data[31:0]	32	Out	Read data bus to the PCU.
DIU Interface			
dnc_diu_rreq	1	Out	DNC unit requests DRAM read. A read request must be accompanied by a valid read address.
dnc_diu_radr[21:5]	17	Out	Read address to DIU, 256-bit word aligned.
diu_dnc_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *dnc_diu_radr*
diu_dnc_rvalid	1	In	Read data valid, active high. Indicates that valid read data is now on the read data bus, *diu_data*.
diu_data[63:0]	64	In	Read data from DIU.
HCU Interface			
dnc_hcu_ready	1	Out	Indicates that DNC is ready to accept data from the HCU.
hcu_dnc_avail	1	In	Indicates valid data present on *hcu_dnc_data*.
hcu_dnc_data[5:0]	6	In	Output bi-level dot data in 6 ink planes.
DWU Interface			
dwu_dnc_ready	1	In	Indicates that DWU is ready to accept data from the DNC.
dnc_dwu_avail	1	Out	Indicates valid data present on *dnc_dwu_data*.
dnc_dwu_data[5:0]	6	Out	Output bi-level dot data in 6 ink planes.

29.5.2   Configuration registers

The configuration registers in the DNC are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for the description of the protocol and timing diagrams for reading and writing registers in the DNC. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the DNC. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *dnc_pcu_data*. Table 150 lists the configuration registers in the DNC.

Table 150. DNC configuration registers

Address DNC_base	Register name	#bits	Value on reset	Description
Control registers				
0x00	Reset	1	0x1	A write to this register causes a reset of the DNC.
0x04	Go	1	0x0	Writing 1 to this register starts the DNC. Writing 0 to this register halts the DNC. When *Go* is asserted all counters, flags etc. are cleared or given their initial value, but configuration registers keep their values. When *Go* is deasserted the state-machines go to their idle states but all counters and configuration registers keep their values. This register can be read to determine if the DNC is running (1 = running, 0 = stopped).
Setup registers (constant during processing)				
0x10	MaxDot	16	0x0000	This is the maximum dot number - 1 present across a page. For example if a page contains 13824 dots, then *MaxDot* will be 13823. Note that this number may or may not be the same as the number of dots across the printhead as some margining may be introduced in the PHI.
0x14	LSFR	32	0x0000_0000	The current value of the LFSR register used as the 32-bit maximum length random bit generator. Users can write to this register to program a seed value for the 32-bit maximum length random bit generator. Must not be all 1s for taps implemented in XNOR form. (It is expected that writing a seed value will not occur during the operation of the LFSR). This LSFR value could also have a possible use as a random source in program code.
0x20	FixativeMask1	6	0x00	Defines the higher priority fixative plane(s). Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = the ink plane contains fixative. 0 = the ink plane does not contain fixative.
0x24	FixativeMask2	6	0x00	Defines the lower priority fixative plane(s). Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. Used only when *FixativeMask1* planes are dead. For each bit: 1 = the ink plane contains fixative. 0 = the ink plane does not contain fixative.
0x28	FixativeRequiredMask	6	0x00	Identifies the ink planes that require fixative. Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = the ink plane requires fixative. 0 = the ink plane does not require fixative (e.g. ink is self-fixing)

Table 150. DNC configuration registers

Address (DNC base +)	Register name	#bits	Value on reset	Description
0x30	DnTableStartAdr	17	0x0_0000	Start address of Dead Nozzle Table in DRAM, specified in 256-bit words.
0x34	DnTableEndAdr	17	0x0_0000	End address of Dead Nozzle Table in DRAM, specified in 256-bit words, i.e. the location containing the last entry in the Dead Nozzle Table. The Dead Nozzle Table should be aligned to a 256-bit boundary, if necessary it can be padded with null entries.
0x40 - 0x54	PlaneReplacePattern[5:0]	6x6	0x00	Defines the ink replacement pattern for each of the 6 ink planes. *PlaneReplacePattern[0]* is the ink replacement pattern for plane 0, *PlaneReplacePattern[1]* is the ink replacement pattern for plane 1, etc. For each 6-bit replacement pattern for a plane, a 1 in any bit positions indicates the alternative ink planes to be used for this plane.
0x58	DiffuseEnable	6	0x3F	Defines whether, after ink replacement, error diffusion is allowed to be performed on each plane. Bit 0 represents the settings for plane 0, bit 1 for plane 1 etc. For each bit: 1 = error diffusion is enabled 0 = error diffusion is disabled
Debug registers (read only)				
0x60	DncOutputDebug	8	N/A	Bit 7 = *dwu_dnc_ready* Bit 6 = *dnc_dwu_avail* Bits 5-0 = *dnc_dwu_data*
0x64	DncReplaceDebug	14	N/A	Bit 13 = *edu_ready* Bit 12 = *iru_avail* Bits 11-6 = *iru_dn_mask* Bits 5-0 = *iru_data*
0x68	DncDiffuseDebug	14	N/A	Bit 13 = *dwu_dnc_ready* Bit 12 = *dnc_dwu_avail* Bits 11-6 = *edu_dn_mask* Bits 5-0 = *edu_data*

29.5.3   Ink replacement unit

Figure 208 shows a sub-block diagram for the ink replacement unit.
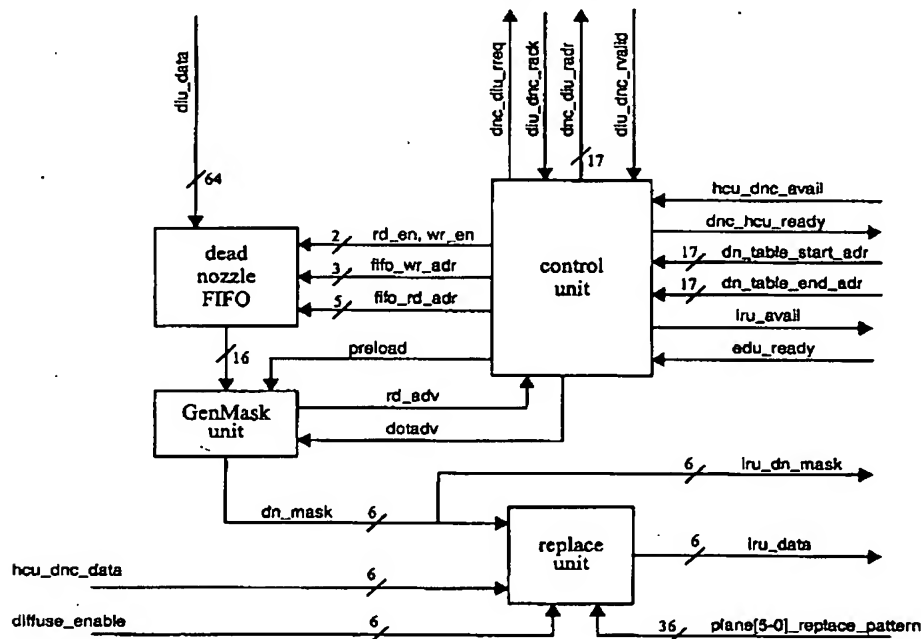

Figure 208. Sub-block diagram of ink replacement unit

29.5.3.1 Control unit

The control unit is responsible for reading the dead nozzle table from DRAM and making it available to the DNC via the dead nozzle FIFO. The dead nozzle table is read from DRAM in single 256-bit accesses, receiving the data from the DIU over 4 clock cycles (64-bits per cycle). The protocol and timing for read accesses to DRAM is described in section 20.9.1 on page 208. Reading from DRAM is implemented by means of the state machine shown in Figure 209.

All counters and flags should be cleared after reset. When *Go* transitions from 0 to 1 all counters and flags should take their initial value. While the *Go* bit is 1, the state machine requests a read access from the dead nozzle table in DRAM provided there is enough space in its FIFO.

A modulo-4 counter, *rd_count*, is used to count each of the 64-bits received in a 256-bit read access. It is incremented whenever *diu_dnc_rvalid* is asserted. When *Go* is 1, *dn_table_radr* is set to *dn_table_start_adr*. As each 64-bit value is returned, indicated by *diu_dnc_rvalid* being asserted, *dn_table_radr* is compared to *dn_table_end_adr*:

* If *rd_count* equals 3 and *dn_table_radr* equals *dn_table_end_adr*, then *dn_table_radr* is updated to *dn_table_start_adr*.
* If *rd_count* equals 3 and *dn_table_radr* does not equal *dn_table_end_adr*, then *dn_table_radr* is incremented by 1.

A count is kept of the number of 64-bit values in the FIFO. When *diu_dnc_rvalid* is 1 data is written to the FIFO by asserting *wr_en*, and *fifo_contents* and *fifo_wr_adr* are both incremented.

When *fifo_contents[3:0]* is greater than 0 and *edu_ready* is 1, *dnc_hcu_ready* is asserted to indicate that the DNC is ready to accept dots from the HCU. If *hcu_dnc_avail* is also 1 then a *dotadv* pulse is sent to the GenMask unit, indicating the DNC has accepted a dot from the HCU, and *iru_avail* is also asserted. After *Go* is set, a single *preload* pulse is sent to the GenMask unit once the FIFO contains data.

When a *rd_adv* pulse is received from the GenMask unit, *fifo_rd_adr[4:0]* is then incremented to select the next 16-bit value. If *fifo_rd_adr[1:0]* = 11 then the next 64-bit value is read from the FIFO by asserting *rd_en*, and *fifo_contents[3:0]* is decremented.
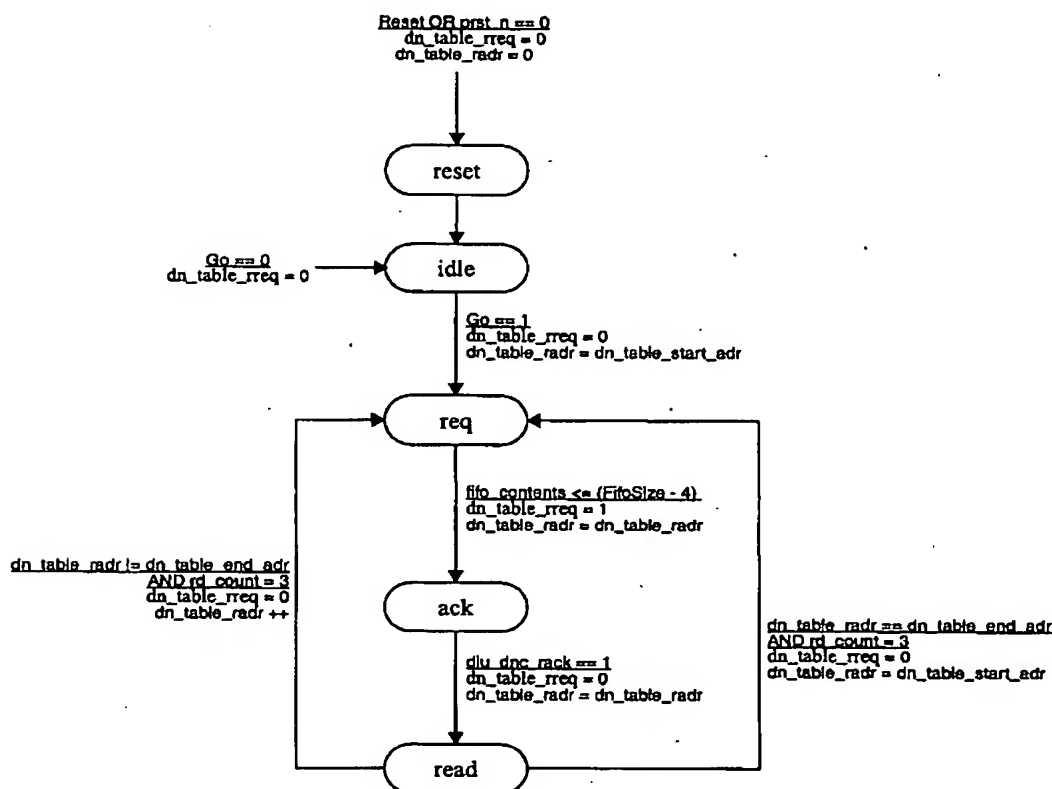

Figure 209. Dead nozzle table state machine

29.5.3.2 Dead nozzle FIFO

The dead nozzle FIFO conceptually is a 64-bit input, and 16-bit output FIFO to account for the 64-bit data transfers from the DIU, and the individual 16-bit entries in the dead nozzle table that are used in the Gen-Mask unit. In reality, the FIFO is actually 8 entries deep and 64-bits wide (to accommodate two 256-bit accesses).

On the DRAM side of the FIFO the write address is 64-bit aligned while on the GenMask side the read address is 16-bit aligned, i.e. the upper 3 bits are input as the read address for the FIFO and the lower 2 bits

are used to select 16 bits from the 64 bits (1st 16 bits read corresponds to bits 15-0, second 16 bits to bits 31-16 etc.).

29.5.3.3 GenMask unit

The GenMask unit generates the 6-bit *dn_mask* that is sent to the replace unit. It consists of a 10-bit delta counter and a mask register.

After *Go* is set, the GenMask unit will receive a *preload* pulse from the control unit indicating the first dead nozzle table entry is available at the output of the dead nozzle FIFO and should be loaded into the delta counter and mask register. A *rd_adv* pulse is generated so that the next dead nozzle table entry is presented at the output of the dead nozzle FIFO. The delta counter is decremented every time a *dotadv* pulse is received. When the delta counter reaches 0, it gets loaded with the current delta value output from the dead nozzle FIFO, i.e. bits 15-6, and the mask register gets loaded with mask output from the dead nozzle FIFO, i.e. bits 5-0. A *rd_adv* pulse is then generated so that the next dead nozzle table entry is presented at the output of the dead nozzle FIFO.

When the delta counter is 0 the value in the mask register is output as the *dn_mask*, otherwise the *dn_mask* is all 0s.

The GenMask unit has no knowledge of the number of dots in a line, it simply loads a counter to count the delta from one dead nozzle column to the next. Thus as described in section 29.2 on page 446 the dead nozzle table should include null identifiers if necessary so that the dead nozzle table covers the first and last nozzle column in a line.

29.5.3.4 Replace unit

Dead nozzle removal and ink replacement are implemented by the combinatorial logic shown in Figure 210. Dead nozzle removal is performed by bit-wise ANDing of the inverse of the *dn_mask* with the dot value.

The ink replacement mechanism has 6 ink replacement patterns, one per ink plane, programmable by the CPU. The dead nozzle mask is ANDed with the dot data to see if there are any planes where the dot is active but the corresponding nozzle is dead. The resultant value forms an enable, on a per ink basis, for the ink replacement process. If replacement is enabled for a particular ink, the values from the corresponding replacement pattern register are ORed into the dot data. The output of the ink replacement process is then filtered so that error diffusion is only allowed for the planes in which error diffusion is enabled.

The output of the ink replacement process is ORed with the resultant dot after dead nozzle removal. If the dot position does not contain a dead nozzle then the *dn_mask* will be all 0s and the dot, *hcu_dnc_data*, will be passed through unchanged.
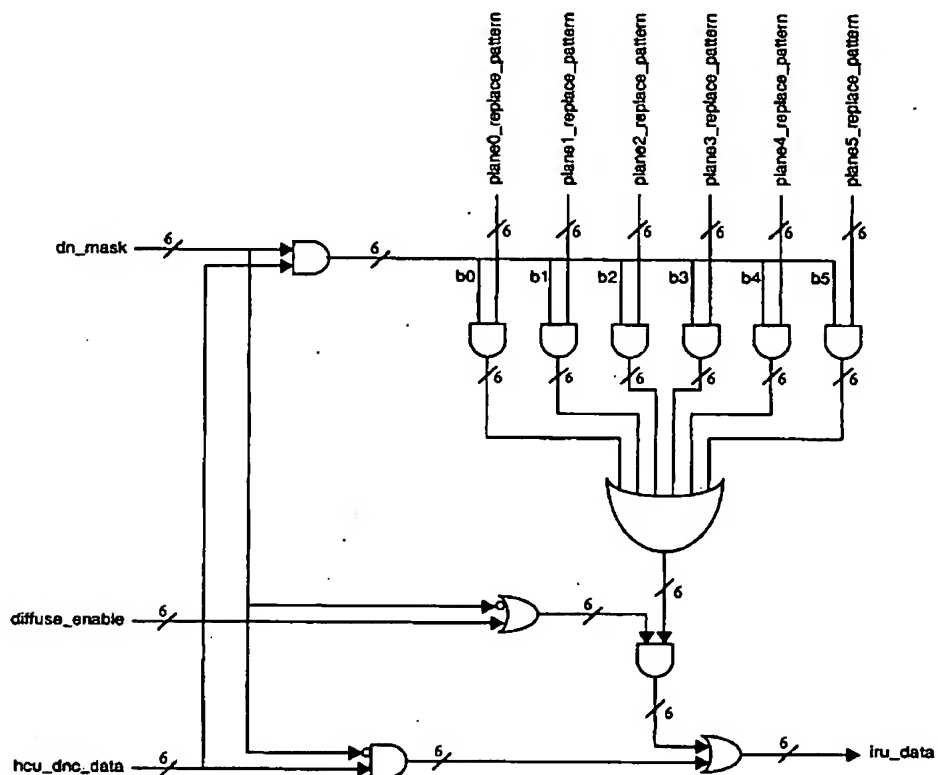
Figure 210. Logic for dead nozzle removal and ink replacement

29.5.4 Error Diffusion Unit

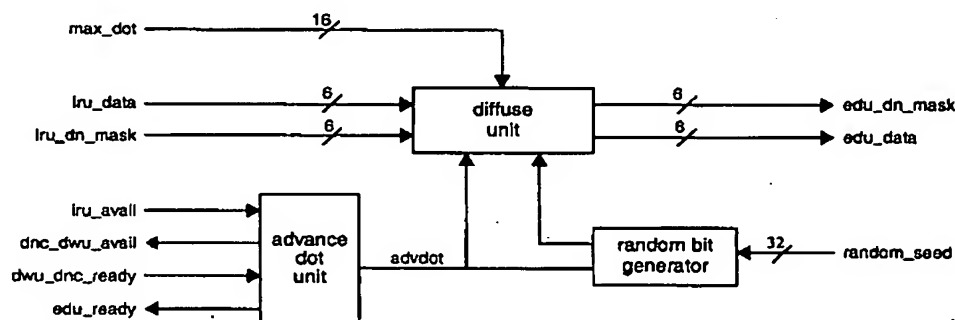Figure 211 shows a sub-block diagram for the error diffusion unit.

Figure 211. Sub-block diagram of error diffusion unit

29.5.4.1 Random Bit Generator

The random bit value used to arbitrarily select the direction of diffusion is generated by a maximum length 32-bit LFSR. The tap points and feedback generation are shown in Figure 212. The LFSR generates a new bit for each dot in a line regardless of whether the dot is dead or not, i.e shifting of the LFSR is enabled when *advdot* equals 1. The LFSR can be initialised with a 32-bit programmable seed value, *random_seed*. This seed value is loaded into the LFSR whenever a write occurs to the *RandomSeed* register. Note that the seed value must not be all 1s as this causes the LFSR to lock-up.
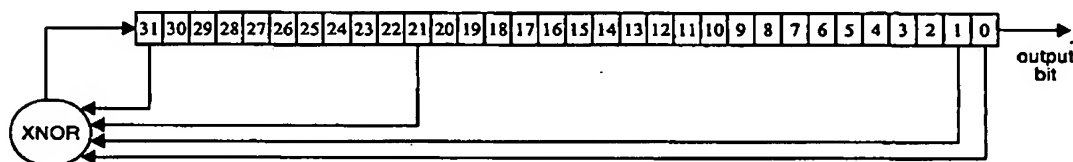

Figure 212. Maximum length 32-bit LFSR used for random bit generation

29.5.4.2 Advance Dot Unit

The advance dot unit is responsible for determining in a given cycle whether or not the error diffuse unit will accept a dot from the ink replacement unit or make a dot available to the fixative correct unit and on to the DWU. It therefore receives the *dwu_dnc_ready* control signal from the DWU, the *iru_avail* flag from the ink replacement unit, and generates *dnc_dwu_avail* and *edu_ready* control flags.

Only the *dwu_dnc_ready* signal needs to be checked to see if a dot can be accepted and asserts *edu_ready* to indicate this. If the error diffuse unit is ready to accept a dot and the ink replacement unit has a dot available, then a *advdot* pulse is given to shift the dot into the pipeline in the diffuse unit. Note that since the error diffusion operates on 3 dots, the advance dot unit ignores *dwu_dnc_ready* initially until 3 dots have been accepted by the diffuse unit. Similarly *dnc_dwu_avail* is not asserted until the diffuse unit contains 3 dots and the ink replacement unit has a dot available.

29.5.4.3 Diffuse Unit

The diffuse unit contains the combinatorial logic to implement the truth table from Table 147. The diffuse unit receives a dot consisting of 6 color planes (1 bit per plane) as well as an associated 6-bit dead nozzle mask value.

Error diffusion is applied to all 6 planes of the dot in parallel. Since error diffusion operates on 3 dots, the diffuse unit has a pipeline of 3 dots and their corresponding dead nozzle mask values. The first dot received is referred to as dot A, and the second as dot B, and the third as dot C. Dots are shifted along the pipeline whenever *advdot* is 1. A count is also kept of the number of dots received. It is incremented whenever *advdot* is 1, and wraps to 0 when it reaches *max_dot*. When the dot count is 0 dot C corresponds to the first dot in a line. When the dot count is 1 dot A corresponds to the last dot in a line.

In any given set of 3 dots only dot B can be defined as containing a dead nozzle(s). Dead nozzles are identified by bits set in *iru_dn_mask*. If dot B contains a dead nozzle(s), the corresponding bit(s) in dot A, dot C, the dead nozzle mask value for A, the dead nozzle mask value for C, the dot count, as well as the random bit value are input to the truth table logic and the dots A, B and C assigned accordingly. If dot B does not contain a dead nozzle then the dots are shifted along the pipeline unchanged.

29.5.5 Fixative Correction Unit

The fixative correction unit consists of combinatorial logic to implement fixative correction as defined in Table 151. For each output dot the DNC determines if fixative is required for the new compensated dot data word and whether fixative is activated already for that dot.

```
FixativePresent = ((FixativeMask1 | FixativeMask2) & edu_data) != 0
FixativeRequired = (FixativeRequiredMask & edu_data) != 0
```

It then looks up the truth table to see what action, if any, needs to be taken.

Table 151. Truth table for fixative correction

Fixative Present	Fixative Required	Action	Output
1	1	Output dot as is.	dnc_dwu_data = edu_data
1	0	Clear fixative plane.	dnc_dwu_data = (edu_data) & ~(FixativeMask1 | FixativeMask2)

0	1	Attempt to add fixative.	if (FixativeMask1 & DnMask) != 0
			  dnc_dwu_data = (edu_data) | (FixativeMask2 & ~DnMask)
			else
			  dnc_dwu_data = (edu_data) | (FixativeMask1)

0	0	Output dot as is.	dnc_dwu_data = edu_data

When attempting to add fixative the DNC first tries to add it into the plane defined by *FixativeMask1*. However, if this plane is dead then it tries to add fixative by placing it into the plane defined by *FixativeMask2*. Note that if both *FixativeMask1* and *FixativeMask2* are both all 0s then the dot data will not be changed.

30 Dotline Writer Unit (DWU)

30.1 OVERVIEW

The Dotline Writer Unit (DWU) receives 1 dot (6 bits) of color information per cycle from the DNC. Dot data received is bundled into 256-bit words and transferred to the DRAM. The DWU (in conjunction with the LLU) implements a dot line FIFO mechanism to compensate for the physical placement of nozzles in a printhead, and provides data rate smoothing to allow for local complexities in the dot data generate pipe-line.
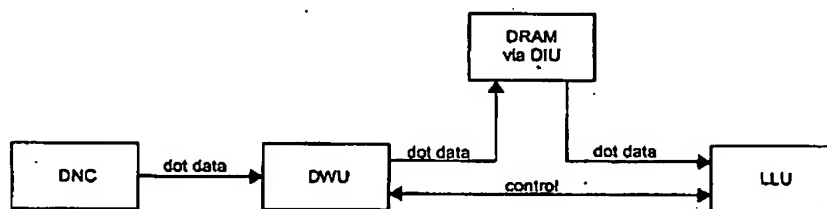
Figure 213. High level data flow diagram of DWU in context

30.2 PHYSICAL REQUIREMENT IMPOSED BY THE PRINTHEAD

The physical placement of nozzles in the printhead means that in one firing sequence of all nozzles, dots will be produced over several print lines. The printhead consists of 12 rows of nozzles, one for each color of odd and even dots. Odd and even nozzles are separated by D_2 print lines and nozzles of different colors are separated by D_1 print lines. See Figure 214 for reference. The first color to be printed is the first row of nozzles encountered by the incoming paper. In the example this is color 0 odd, although is dependent on

the printhead type (see Section 35 Memjet Printhead for other printhead arrangments). Paper passes under printhead moving downwards.

Figure 214. Printhead Nozzle Layout for conceptual 36 Nozzle bi-lithic printhead

For example if the physical separation of each half row is $80\mu m$ equating to $D_1=D_2=5$ print lines at 1600dpi. This means that in one firing sequence, color 0 odd nozzles will fire on dotline L, color 0 even nozzles will fire on dotline $L-D_1$, color 1 odd nozzles will fire on dotline $L-D_1-D_2$ and so on over 6 color

planes odd and even nozzles. The total number of lines fired over is given as 0+5+5.....+5= 0 + 11x5 =55. See Figure 215 for example diagram.
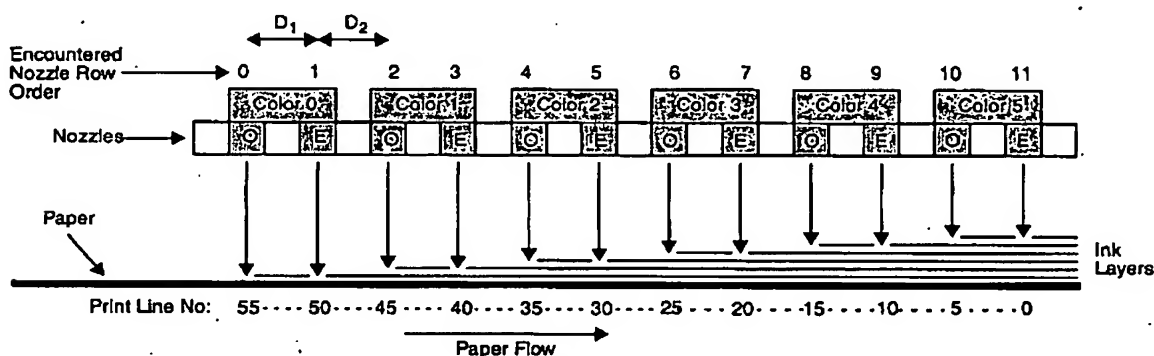

Figure 215. Paper and printhead nozzles relationship (example with D$_1$=D$_2$=5)

It is expected that the physical spacing of the printhead nozzles will be 80µm (or 5 dot lines), although there is no dependency on nozzle spacing. The DWU is configurable to allow other line nozzle spacings.

Table 152. Relationship between Nozzle color/sense and line firing

Color	Even line encountered first		Odd line encountered first	
	sense	line	sense	line
Color 0	even	L	even	L-5
	odd	L-5	odd	L
Color 1	even	L-10	even	L-15
	odd	L-15	odd	L-10
Color 2	even	L-20	even	L-25
	odd	L-25	odd	L-20
Color 3	even	L-30	even	L-35
	odd	L-35	odd	L-30
Color 4	even	L-40	even	L-45
	odd	L-45	odd	L-40
Color 5	even	L-50	even	L-55
	odd	L-55	odd	L-50

30.3   LINE RATE DE-COUPLING

The DWU block is required to compensate for the physical spacing between lines of nozzles. It does this by storing dot lines in a FIFO (in DRAM) until such time as they are required by the LLU for dot data transfer to the printhead interface. Colors are stored separately because they are needed at different times by the LLU. The dot line store must store enough lines to compensate for the physical line separation of the printhead but can optionally store more lines to allow system level data rate variation between the read (printhead feed) and write sides (dot data generation pipeline) of the FIFOs.

A logical representation of the FIFOs is shown in Figure 216, where N is defined as the optional number of extra half lines in the dot line store for data rate de-coupling.

Even Row Encountered First

Odd Row Encountered First
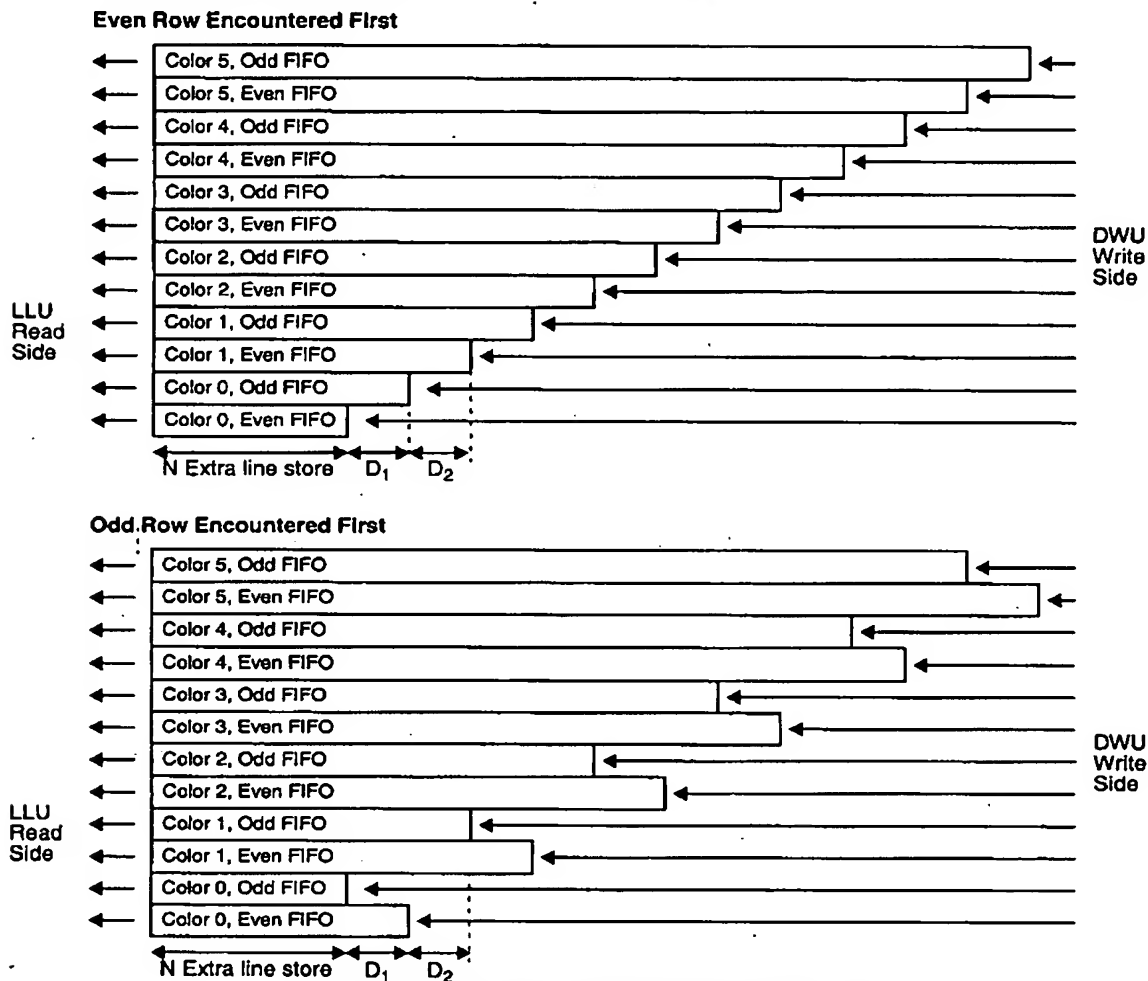
Figure 216. Dot line store logical representation

30.4 DOT LINE STORE STORAGE REQUIREMENTS

For an arbitrary page width of d dots (where d is even), the number of dots per half line is $d/2$.

For interline spacing of D_2 and inter-color spacing of D_1, with C colors of odd and even half lines, the number of half line storage is $(C - 1)(D_2+D_1) + D1$.

For N extra half line stores for each color odd and even, the storage is given by $(N * C * 2)$.

The total storage requirement is $((C - 1)(D_2+D_1) + D1 + (N * C * 2)) * d/2$ in bits.

Note that when determining the storage requirements for the dot line store, the number of dots per line is the page width and not necessarily the printhead width. The page width is often the dot margin number of dots less than the printhead width. They can be the same size for full bleed printing.

For example in an A4 page a line consists of 13824 dots at 1600 dpi, or 6912 dots per half dot line. To store just enough dot lines to account for an inter-line nozzle spacing of 5 dot lines it would take 55 half dot lines for color 5 odd, 50 dot lines for color 5 even and so on, giving 55+50+45...10+5+0= 330 half dot lines in total. If it is assumed that N=4 then the storage required to store 4 extra half lines per color is 4 x 12=48, in total giving 330+48=378 half dot lines. Each half dot line is 6912 dots, at 1 bit per dot give a total storage requirement of 6912 dots x 378 half dot lines / 8 bits = Approx 319 Kbytes. Similarly for an A3 size page with 19488 dots per line, 9744 dots per half line x 378 half dot lines / 8 = Approx 899 Kbytes.

Table 153. Storage requirement for dot line store

Page size	Nozzle Spacing	Lines required (N=0)	Storage (N=0) Kbytes	Lines required (N=4)	Storage (N=4) Kbytes
A4	4	264	223	312	263
	5	330	278	378	319
A3	4	264	628	312	742
	5	330	785	378	899

The potential size of the dot line store makes it unfeasible to be implemented in on-chip SRAM, requiring the dot line store to be implemented in embedded DRAM. This allows a configurable dotline store where unused storage can be redistributed for use by other parts of the system.

30.5 LOCAL BUFFERING

An embedded DRAM is expected to be of the order of 256 bits wide, which results in 27 words per half line of an A4 page, and 54 words per half line of A3. This requires 27 words x 12 half colors (6 colors odd and even) = 324 x 256-bit DRAM accesses over a dotline print time, equating to 6 bits per cycle (equal to DNC generate rate of 6 bits per cycle). Each half color is required to be double buffered, while filling one buffer the other buffer is being written to DRAM. This results in 256 bits x 2 buffers x 12 half colors i.e. 6144 bits in total.

The buffer requirement can be reduced, by using 1.5 buffering, where the DWU is filling 128 bits while the remaining 256 bits are being written to DRAM. While this reduces the required buffering locally it increases the peak bandwidth requirement to the DRAM. With 2x buffering the average and peak DRAM bandwidth requirement is the same and is 6 bits per cycle, alternatively with 1.5x buffering the average DRAM bandwidth requirement is 6 bits per cycle but the peak bandwidth requirement is 12 bits per cycle. The amount of buffering used will depend on the DRAM bandwidth available to the DWU unit.
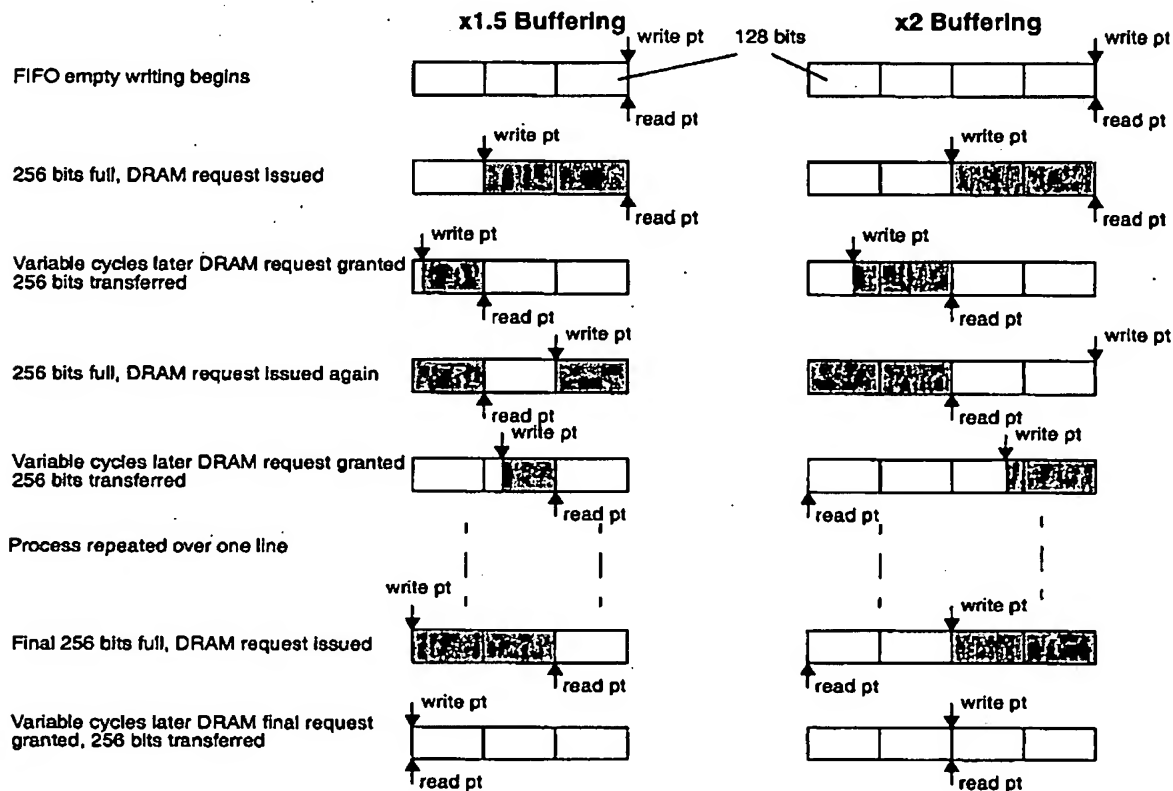
x1.5 Buffering   **x2 Buffering**

FIFO empty writing begins

256 bits full, DRAM request issued

Variable cycles later DRAM request granted
256 bits transferred

256 bits full, DRAM request issued again

Variable cycles later DRAM request granted
256 bits transferred

Process repeated over one line

Final 256 bits full, DRAM request issued

Variable cycles later DRAM final request
granted, 256 bits transferred

Figure 217. Comparison of 1.5x v 2x buffering

Should the DWU fail to get the required DRAM access within the specified time, the DWU will stall the DNC data generation. The DWU will issue the stall in sufficient time for the DNC to respond and still not cause a FIFO overrun. Should the stall persist for a sufficiently long time, the PHI will be starved of data and be unable to deliver data to the printhead in time. The sizing of the dotline store FIFO and internal FIFOs should be chosen so as to prevent such a stall happening.

30.6 DOTLINE DATA IN MEMORY

The dot data shift register order in the printhead is shown in Figure 214 (the transmit order is the opposite of the shift register order). In the example the type 0 printhead IC transmit order is increasing even color data followed by decreasing odd color data. The type 1 printhead IC transmit order is decreasing odd color data followed by increasing even color data. For both printhead ICs the even data is always increasing order and odd data is always decreasing. The PHI controls which printhead IC data gets shifted to.

From this it is beneficial to store even data in increasing order in DRAM and odd data in decreasing order. While this order suits the example printhead, other printheads exist where it would be beneficial to store even data in decreasing order, and odd data in increasing order, hence the order is configurable. The order that data is stored in memory is controlled by setting the *ColorLineSense* register.

The dot order in DRAM for increasing and decreasing sense is shown in Figure 218 and Figure 219 respectively. For each line in the dot store the order is the same (although for odd lines the numbering will be different the order will remain the same). Dot data from the DNC is always received in increasing dot number order. For increasing sense dot data is bundled into 256-bit words and written in increasing order in DRAM, word 0 first, then word 1, and so on to word N, where N is the number of words in a line.

For decreasing sense dot data is also bundled into 256-bit words, but is written to DRAM in decreasing order, i.e. word N is written first then word N-1 and so on to word 0. For both increasing and decreasing sense the data is aligned to bit 0 of a word, i.e. increasing sense always starts at bit 0, decreasing sense always finishes at bit 0.
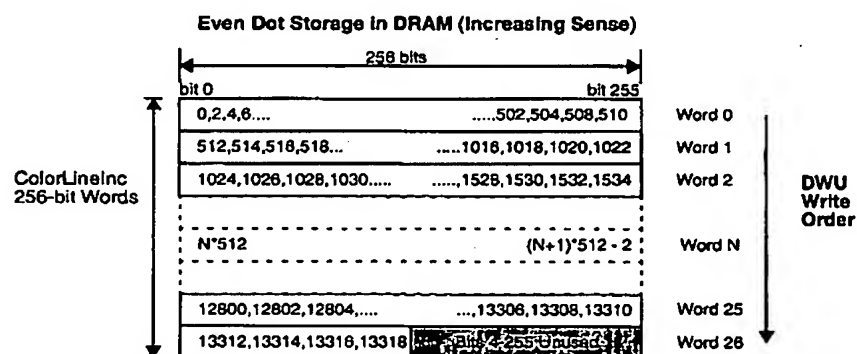
Even Dot Storage in DRAM (Increasing Sense)

Figure 218. Even dot order in DRAM (Increasing Sense, 13320 dot wide line)

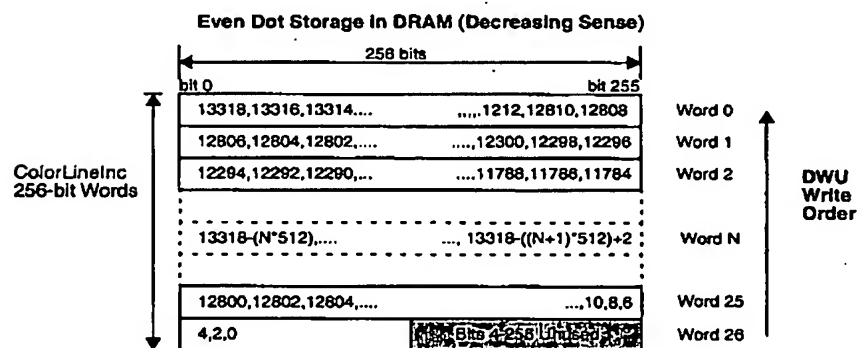**Even Dot Storage in DRAM (Decreasing Sense)**

Figure 219. Even dot order in DRAM (Decreasing Sense, 13320 dot wide line)

Each half color is configured independently of any other color. The *ColorBaseAdr* register specifies the position where data for a particular dotline FIFO will begin writing to. Note that for increasing sense colors the *ColorBaseAdr* register specifies the address of the first word of first line of the fifo, whereas for decreasing sense colors the *ColorBaseAdr* register specifies the address of last word of the first line of the FIFO.

Dot data received from the DNC is bundled in 256-bit words and transferred to the DRAM. Each line of data is stored consecutively in DRAM, with each line separated by *ColorLineInc* number of words.

For each line stored in DRAM the DWU increments the line count and calculates the DRAM address for the next line to store.

This process continues until *ColorFifoSize* number of lines are stored, after which the DRAM address with wrap back to the *ColorBaseAdr* address.
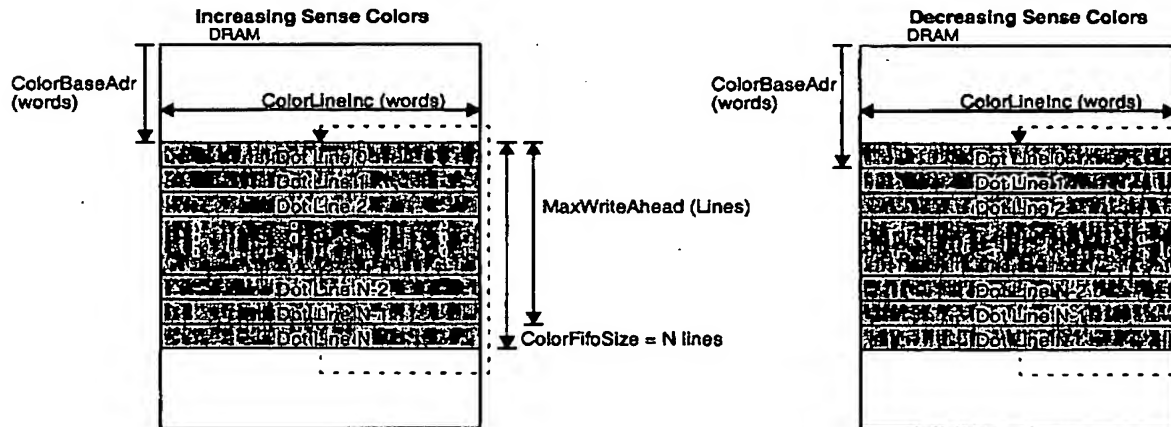

Figure 220. Dotline FIFO data structure in DRAM

As each line is written to the FIFO, the DWU increments the *FifoFillLevel* register, and as the LLU reads a line from the FIFO the *FifoFillLevel* register is decremented. The LLU indicates that it has completed reading a line by a high pulse on the *llu_dwu_line_rd* line.

When the number of lines stored in the FIFO is equal to the *MaxWriteAhead* value the DWU will indicate to the DNC that it is no longer able to receive data (i.e. a stall) by deasserting the *dwu_dnc_ready* signal.

The *ColorEnable* register determines which color planes should be processed, if a plane is turned off, data is ignored for that plane and no DRAM accesses for that plane are generated.

30.7   IMPLEMENTATION

30.7.1   Definitions of I/O

Table 154. DWU I/O Definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
prst_n	1	In	System reset, synchronous active low
DNC Interface			
dwu_dnc_ready	1	Out	Indicates that DWU is ready to accept data from the DNC.
dnc_dwu_avail	1	In	Indicates valid data present on *dnc_dwu_data*.
dnc_dwu_data[5:0]	6	In	Input bi-level dot data in 6 ink planes.
LLU Interface			
dwu_llu_line_wr	1	Out	DWU line write. Indicates that the DWU has completed a full line write. Active high
llfu_dwu_line_rd	1	In	LLU line read. Indicates that the LLU has completed a line read. Active high.
LLU and DWU common configuration			
dwu_llu_cfifosize[11:0][7:0]	12x8	Out	Indicates the number of lines in the FIFO before the line increment will wrap around in memory. Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5
PCU Interface			
pcu_dwu_sel	1	In	Block select from the PCU. When *pcu_dwu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
dwu_pcu_rdy	1	Out	Ready signal to the PCU. When *dwu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *dwu_pcu_data* is valid.
dwu_pcu_data[31:0]	32	Out	Read data bus to the PCU.
DIU Interface			
dwu_diu_wreq	1	Out	DWU requests DRAM write. A write request must be accompanied by a valid write address together with valid write data and a write valid.
dwu_diu_wadr[21:5]	17	Out	Write address to DIU 17 bits wide (256-bit aligned word)
diu_dwu_wack	1	In	Acknowledge from DIU that write request has been accepted and new write address can be placed on *dwu_diu_wadr*

Table 154. DWU I/O Definition

Port name	Bits	I/O	Description
dwu_diu_data[63:0]	64	Out	Data from DWU to DIU. 256-bit word transfer over 4 cycles First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
dwu_diu_wvalid	1	Out	Signal from DWU indicating that data on *dwu_diu_data* is valid.

30.7.2 DWU partition
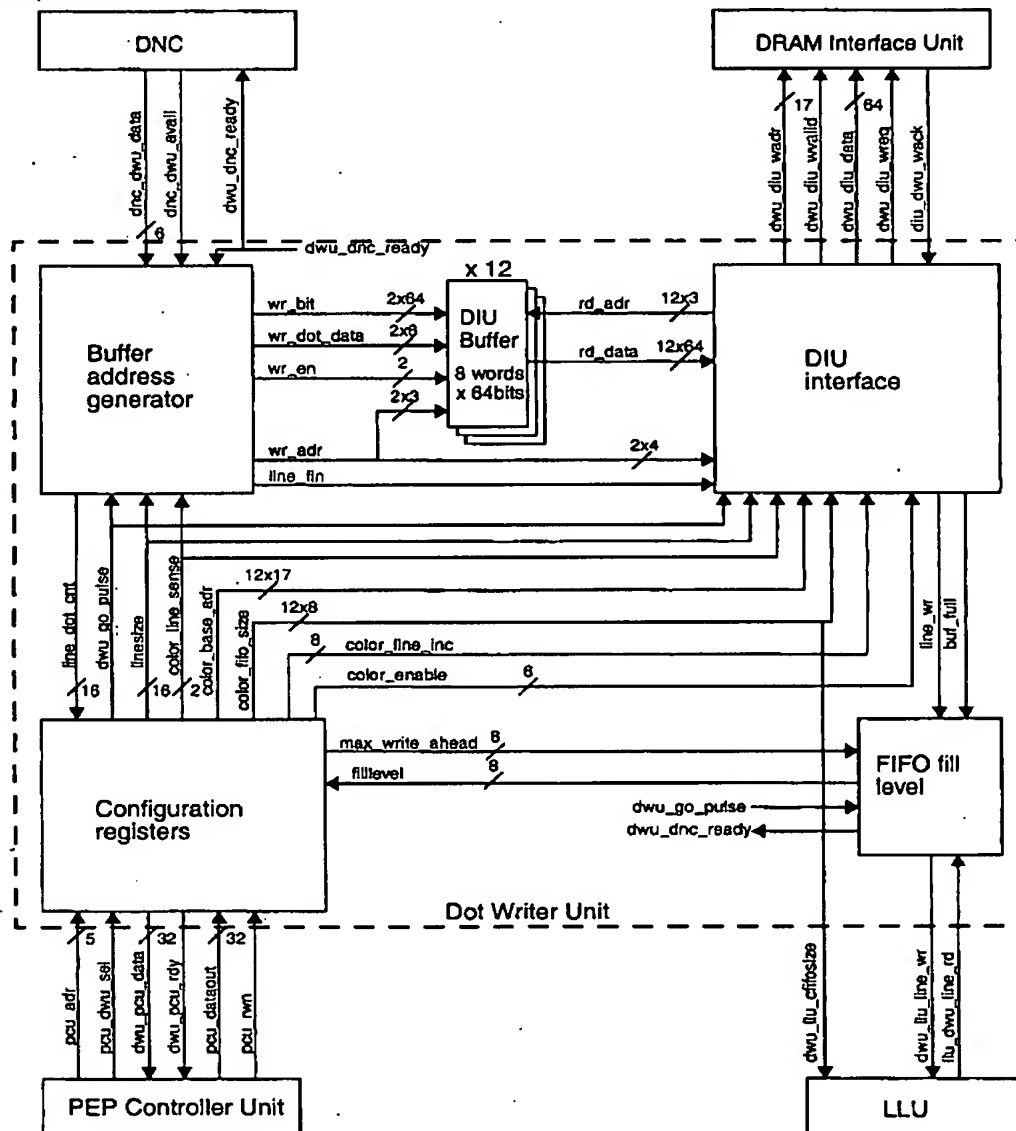
Figure 221. DWU partition

30.7.3 Configuration registers

The configuration registers in the DWU are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for a description of the protocol and timing diagrams for reading and writing registers in the DWU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for

the DWU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *dwu_pcu_data*. Table 155 lists the configuration registers in the DWU.

Table 155. DWU registers description

Address DWU base	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a DWU block reset.
0x04	Go	1	0x0	Active high bit indicating the DWU is programmed and ready to use. A low to high transition will cause DWU block internal states to reset (configuration registers are not reset).
Dot Line Store Configuration				
0x08 - 0x38	ColorBaseAdr[11:0]	12x17	0x00000	Specifies the base address (in words) in memory where data from a particular half color (N) will be placed.
0x3C - 0x6C	ColorFifoSize[11:0]	12x8	0x00	Indicates the number of lines in the FIFO before the line increment will wrap around in memory. Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5
0x70	ColorLineSense	2	0x2	Specifies whether data written to DRAM for this half color is increasing or decreasing sense 0 - Decreasing sense 1 - Increasing sense Bit 0 Defines even color sense, Bit 1 Defines odd color sense.
0x74	ColorEnable	6	0x3F	Indicates whether a particular color is active or not. When inactive no data is written to DRAM for that color. 0 - Color off 1 - Color on One bit per color, bit 0 is Color 0 and so on.
0x78	MaxWriteAhead	8	0x00	Specifies the maximum number of lines that the DWU can be ahead of the LLU
0x7C	LineSize	16	0x0000	Indicates the number of dots per line.
Working Registers				
0x80	LineDotCnt	16	0x0000	Indicates the number of remaining dots in the current line. (Read Only)
0x84	FifoFillLevel	8	0x00	Number of lines in the FIFO, written to but not read. (Read Only)

A low to high transition of the *Go* register causes the internal states of the DWU to be reset. All configuration registers will remain the same. The block indicates the transition to other blocks via the *dwu_go_pulse* signal.

The *ColorLineInc* bus specifies the number of addresses (in 256-bit words) between successive half lines in the dot line store. It is derived from the *LineSize* register by rounding up the nearest 256-bit value. The same value used for all half colors.

```
if (line_size[7:0] !=0 ) then
    color_line_inc[7:0] = line_size[15:8] + 1
```

```
        else
           color_line_inc[7:0] = line_size[15:8];
```

30.7.4 Fifo fill level

The DWU keeps a running total of the number of lines in the dot store FIFO. Each time the DWU writes a line to DRAM (determined by the DIU interface subblock and signalled via *line_wr*) it increments the *filllevel* and signals the line increment to the LLU (pulse on *dwu_llu_line_wr*). Conversely if it receives an active *llu_dwu_line_rd* pulse from the LLU, the *filllevel* is decremented. If the *filllevel* increases to the programmed max level (*max_write_ahead*) then the DWU stalls and indicates back to the DNC by de-asserting the *dwu_dnc_ready* signal.

If one or more of the DIU buffers fill, the DIU interface signals the fill level logic via the *buf_full* signal which in turn causes the DWU to de-assert the *dwu_dnc_ready* signal to stall the DNC. The *buf_full* signals will remain active until the DIU services a pending request from the full buffer, reducing the buffer level.

The DWU does not increment the fill level until a complete line of dot data is in DRAM not just a complete line received from the DNC. This ensures that the LLU cannot start reading a partial line from DRAM before the DWU has finished writing the line.

The fill level is reset to zero each time a new page is started, on receiving a pulse via the *dwu_go_pulse* signal.

The line fifo fill level can be read by the CPU via the PCU at any time by accessing the *FifoFillLevel* register.
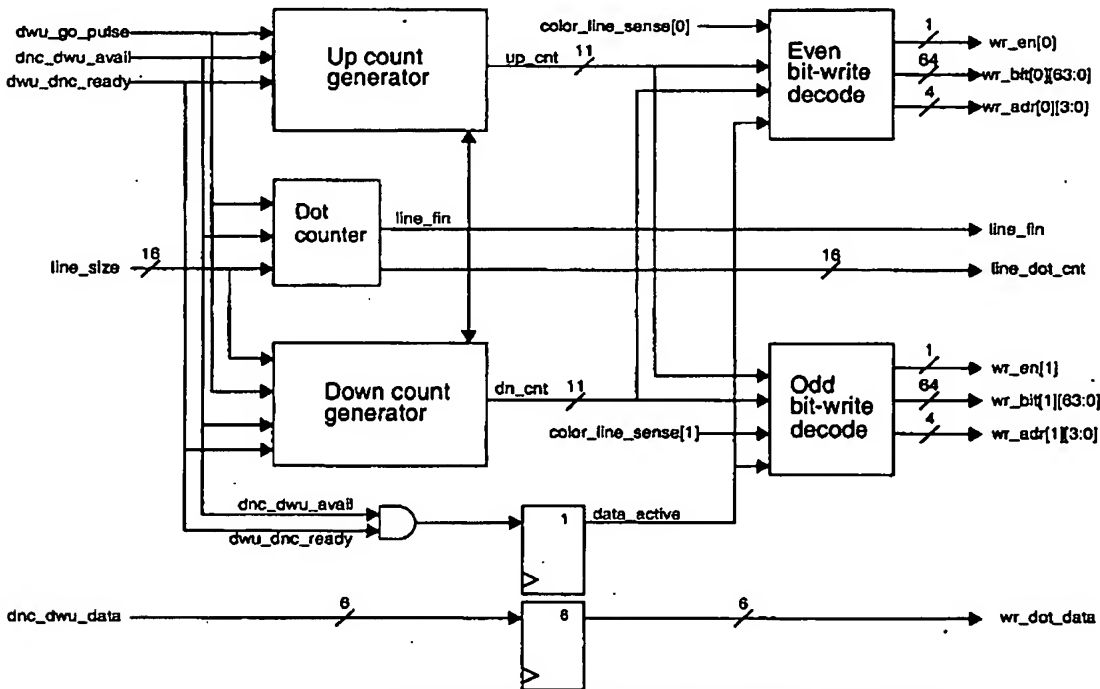
30.7.5 Buffer address generator

Figure 222. Buffer address generator sub-block

30.7.5.1 Buffer address generator description

The buffer address generator subblock is responsible for accepting data from the DNC and writing it to the DIU buffers in the correct order.

The buffer address and active bit-write for a particular dot data write is calculated by the buffer address generator based on the dot count of the current line, programmed sense of the color and the line size.

All configuration registers should be programmed while the *Go* bit is set to zero, once complete the block can be enabled by setting the *Go* bit to one. The transition from zero to one will cause the internal states to reset.

If the *color_line_sense* signal for a color is one (i.e. increasing) then the bit-write generation is straight forward as dot data is aligned with a 256-bit boundary. So for the first dot in that color, the bit 0 of the *wr_bit* bus will be active (in buffer word 0), for the second dot bit 1 is active and so on to the 255th dot where bit 63 is active (in buffer word 3). This is repeated for all 256-bit words until the final word where only a partial number of bits are written before the word is transferred to DRAM.

If *color_line_sense* signal for a color is zero (i.e. decreasing) the bit-write generation for that color is adjusted by an offset calculated from the pre-programmed line length (*line_size*). The offset adjusts the bit write to allow the line to finish on a 256-bit boundary. For example if the line length was 400, for the first dot received bit 7 (line length is halved because of odd/even lines of color) of the *wr_bit* is active (buffer word 3), the second bit 6 (buffer word 3), to the 200th dot of data with bit 0 of *wr_bit* active (buffer word 0).

30.7.5.2 Bit-write decode

The buffer address generator contains 2 instances of the bit-write decode, one configured for odd dot data the other for even. The counter (either up or down counter) used to generate the addresses is selected by the *color_line_sense* signal. Each block determines if it is active on this cycle by comparing its configured type with the current dot count address and the *data_active* signal.

The *wr_bit* bus is a direct decoding of the lower 6 count bits (*count[6:1]*), and the DIU buffer address is the remaining higher bits of the counter (*count[10:7]*).

The signal generation is given as follows:
```
// determine the counter to use
if (color_line_sense == 1 )
   count = up_cnt[10:0]
else
   count = dn_cnt[10:0]
// determine if active, based on instance type
wr_en           = data_active & (count[0] ^ odd_even_type)    // odd =1, even =0
// determine the bit write value
wr_bit[63:0]    = decode(count[6:1])
// determine the buffer 64-bit address
wr_adr[3:0]     = count[10:7]
```

30.7.5.3 Up counter generator

The up counter increments for each new dot and is used to determine the write position of the dot in the DIU buffers for increasing sense data. At the end of each line of dot data (as indicated by *line_fin*), the counter is rounded up to the nearest 256-bit word boundary. This causes the DIU buffers to be flushed to DRAM including any partially filled 256-bit words. The counter is reset to zero if the *dwu_go_pulse* is one.

```
// Up-Counter Logic
if (dwu_go_pulse == 1) then (
   up_cnt[10:0] = 0
elsif (line_fin == 1 ) then
   // round up
   if (up_cnt[8:1] != 0)
      up_cnt[10:9]++
   else
      up_cnt[10:9]
   // bit-selector
   up_cnt[7:0]=0
elsif ((dnc_dwu_avail == 1) AND (dwu_dnc_ready == 1 )) then
   up_cnt[7:0]++
```

30.7.5.4 Down counter generator

The down counter logic decrements for each new dot and is used to determine the write position of the dot in the DUI buffers for decreasing sense data. When the *dwu_go_pulse* bit is one the lower bits (i.e. 8 to 0) of the counter are reset to line size value (*line_size*), and the higher bits to zero. The bits used to determine the bit-write values and 64-bit word addresses in the DIU buffers begin at line size and count down to zero. The remaining higher bits are used to determine the DIU buffer 256-bit address and buffer fill level, begin at zero and count up. The counter is active when valid dot data is present, i.e. *dnc_dwu_avail* equals 1.

When the end of line is detected (*line_fin* equals 1) the counter is rounded to the next 256-bit word, and the lower bits are reset to the line size value.
```
//Down-Counter Logic
if (dwu_go_pulse == 1) then
```

```
      dn_cnt[8:0]  = line_size[8:0]
      dn_cnt[10:9] = 0
  elsif (line_fin == 1 ) then
     // perform rounding up
     if (dn_cnt[8:1] != 0)
        dn_cnt[10:9]++
     else
        dn_cnt[10:9]
     // bit-select is reset
     dn_cnt[8:0]=line_size[8:0]    // bit select bits
  elsif ((dnc_dwu_avail == 1) AND (dwu_dnc_ready == 1 )) then
     dn_cnt[8:0] --
     dn_cnt[10:9]++
```

30.7.5.5 Dot counter

The dot counter simply counts each active dot received from the DNC. It sets the counter to *line_size* and decrements each time a valid dot is received. When the count equals zero the *line_fin* signal is pulsed and the counter is reset to *line_size*.

The counter is reset to *line_size* when *dwu_go_pulse* is 1.

30.7.6   DIU buffer

The DIU buffer is a 64 bit x 8 word dual port register array with bit write capability. The buffer could be implemented with flip-flops should it prove more efficient.
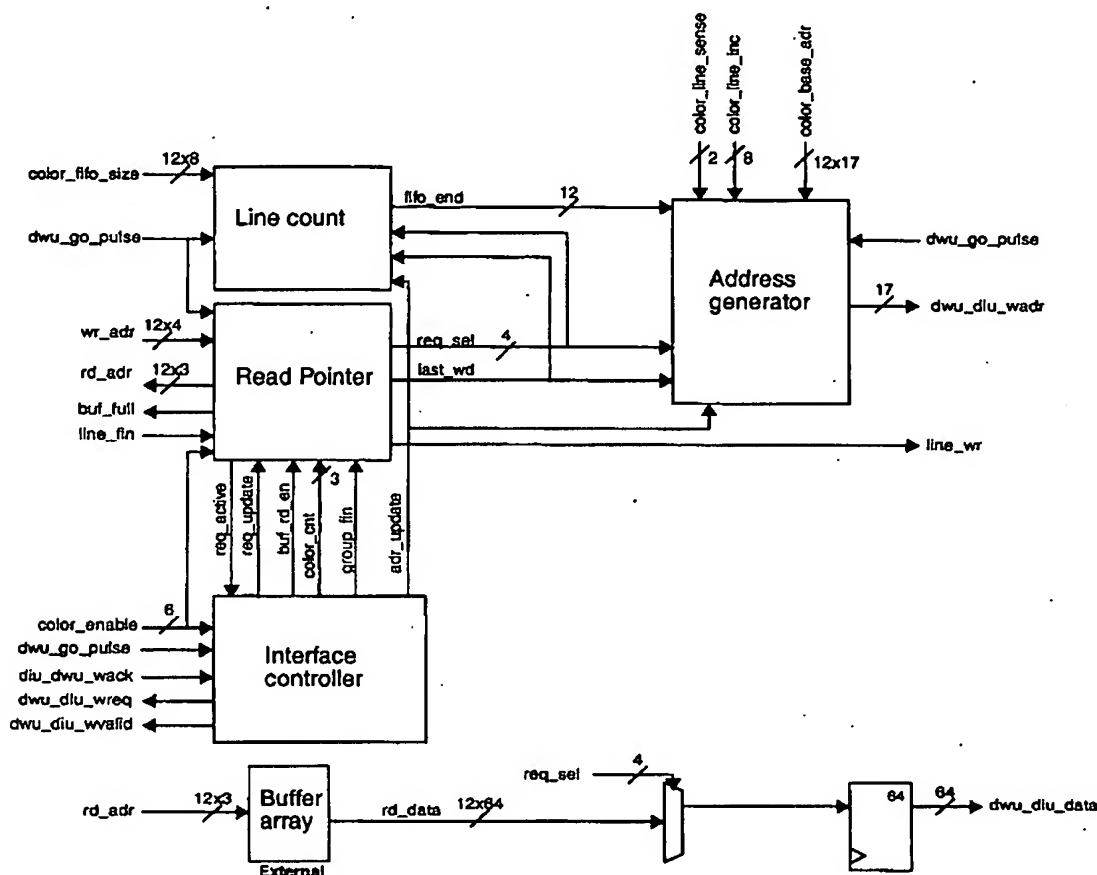
30.7.7   DIU interface

Figure 223. DIU Interface sub-block

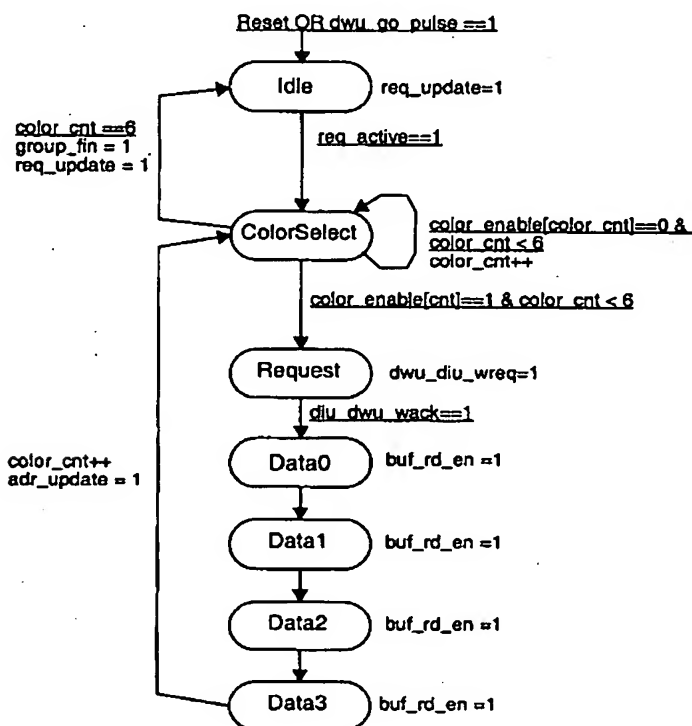#### 30.7.7.1 DIU Interface general description

The DIU interface determines when a buffer needs a data word to be transferred to DRAM. It generates the DRAM address based on the dot line position, the color base address and the other programmed parameters. A write request is made to DRAM and when acknowledged a 256-bit data word is transferred. The interface determines if further words need to be transferred and repeats the transfer process.

If the FIFO in DRAM has reached its maximum level, or one of the buffers has temporarily filled, the DWU will stall data generation from the DNC.

A similar process is repeated for each line until the end of page is reached. At the end of a page the CPU is required to reset the internal state of the block before the next page can be printed. A low to high transition of the *Go* register will cause the internal block reset, which causes all registers in the block to reset with the exception of the configuration registers. The transition is indicated to subblocks by a pulse on *dwu_go_pulse* signal.

30.7.7.2 Interface controller

Figure 224. Interface controller state diagram

The interface controller state machine waits in *Idle* state until an active request is indicated by the read pointer (via the *req_active* signal). When an active request is received the machine proceeds to the *ColorSelect* state to determine which buffers need a data transfer. In the *ColorSelect* state it cycles through each color and determines if the color is enabled (and consequently the buffer needs servicing), if enabled it jumps to the *Request* state, otherwise the *color_cnt* is incremented and the next color is checked.

In the *Request* state the machine issues a write request to the DIU and waits in the *Request* state until the write request is acknowledged by the DIU (*diu_dwu_wack*). Once an acknowledge is received the state machine clocks through 4 cycles transferring 64-bit data words each cycle and incrementing the corresponding buffer read address. After transferring the data to the DIU the machine returns to the *ColorSelect* state to determine if further buffers need servicing. On the transition the controller indicates to the address generator (*adr_update*) to update the address for that selected color.

If all colors are transferred (*color_cnt* equal to 6) the state machine returns to *Idle*, updating the last word flags (*group_fin*) and request logic (*req_update*).

The *dwu_diu_wvalid* signal is a delayed version of the *buf_rd_en* signal to allow for pipeline delays between data leaving the buffer and being clocked through to the DIU block.

The state machine will return from any state to *Idle* if the reset or the *dwu_go_pulse* is 1.

30.7.7.3 Address generator

The address generator block maintains 12 pointers (*color_adr[11:0]*) to DRAM corresponding to current write address in the dot line store for each half color. When a DRAM transfer occurs the address pointer is used first and then updated for the next transfer for that color. The pointer used is selected by the *req_sel* bus, and the pointer update is initiated by the *adr_update* signal from the interface controller.

The pointer update is dependent on the sense of the color of that pointer, the pointer position in a line and the line position in the FIFO. The programming of the *color_base_adr* needs to be adjusted depending of the sense of the colors. For increasing sense colors the *color_base_adr* specifies the address of the first word of first line of the fifo, whereas for decreasing sense colors the *color_base_adr* specifies the address of last word of the first line of the FIFO.

For increasing colors, the initialization value (i.e. when *dwu_go_pulse* is 1) is the *color_base_adr*. For each word that is written to DRAM the pointer in incremented. If the word is the last word in a line (as indicated by *last_wd* from that read pointers) the pointer is also incremented. If the word is the last word in a line, and the line is the last line in the FIFO (indicated by *fifo_end* from the line counter) the pointer is reset to *color_base_adr*.

In the case of decreasing sense colors, the initialization value (i.e. when *dwu_go_pulse* is 1) is the *color_base_adr*. For each line of decreasing sense color data the pointer starts at the line end and decrements to the line start. For each word that is written to DRAM the pointer is decremented. If the word is the last word in a line the pointer is incremented by *color_line_inc* * 2 + 1. One line length to account for the line of data just written, and another line length for the next line to be written. If the word is the last word in a line, and the line is the last line in the FIFO the pointer is reset to the initialization value (i.e. *color_base_adr*).

The address is calculated as follows:

```
if (dwu_go_pulse == 1) then
   color_adr[11:0] = color_base_adr[11:0][21:5]
elsif (adr_update == 1) then {
   // determine the color
   color = req_sel[3:0]
   // line end and fifo wrap
   if ((fifo_end[color] == 1) AND (last_wd == 1)) then {
      // line end and fifo wrap
      color_adr[color] = color_base_adr[color][21:5]
   }
   elsif ( last_wd == 1) then {
      // just a line end no fifo wrap
      if (color_line_sense[color % 2] == 1) then  // increasing sense
         color_adr[color] ++
      else                                        // decreasing sense
         color_adr[color] = color_adr[color] + ( color_line_inc * 2) + 1
   }
   else {
      // regular word write
      if (color_line_sense[color % 2] == 1) then  // increasing sense
         color_adr[color]++
      else                                        // decreasing sense
         color_adr[color]--
   }
}
// select the correct address, for this transfer
dwu_diu_wadr = color_adr[req_sel]
```

30.7.7.4 Line count

The line counter logic counts the number of dot data lines stored in DRAM for each color. A separate pointer is maintained for each color. A line pointer is updated each time the final word of a line is transferred to DRAM. This is determined by a combination of *adr_update* and *last_wd* signals. The pointer to update is indicated by the *req_sel* bus.

When an update occurs to a pointer it is compared to zero, if it is non-zero the count is decremented, otherwise the counter is reset to *color_fifo_size*. If a counter is zero the *fifo_end* signals is set high to indicates to the address generator block that the line is the last line of this colors fifo.

If the *dwu_go_pulse* signal is one the counters are reset to *color_fifo_size*.

```
if (dwu_go_pulse == 1) then
    line_cnt[11:0] = color_fifo_size[11:0]
elsif ((adr_update == 1) AND (last_wd == 1)) then {
    // determine the pointer to operate on
    color = req_sel[3:0]
    // update the pointer
    if (line_cnt[color] == 0) then
        line_cnt[color] = color_fifo_size[color]
    else
        line_cnt[i] --
    }
// count is zero its the last line of fifo
for(i=0 ;i <12;i++){
    fifo_end[i] = (line_cnt[i] == 0)
    }
```

30.7.7.5 Read Pointer

The read pointer logic maintains the buffer read address pointers. The read pointer is used to determine which 64-bit words to read from the buffer for transfer to DRAM.

The read pointer logic compares the read and write pointers of each DIU buffer to determine which buffers require data to be transferred to DRAM (*pend[11:0]* bus), and which buffers are full (the *buf_full* signal). Only enabled buffers are considered as indicated by the *color_enable* bus.

Buffers are grouped into odd and even buffers groups. If an odd buffer requires DRAM access the *odd_pend* signals will be active, if an even buffer requires DRAM access the *even_pend* signals will be active. If both odd and even buffers require DRAM access, the even buffers will get serviced first.

If any buffer requires a DRAM transfer, the logic will indicate to the interface controller via the *req_active* signal, with the *odd_even_sel* signal determining which group of buffers get serviced. The interface controller will check the *color_enable* signal and issue DRAM transfers for all enabled colors in a group. When the transfers are complete it tells the read pointer logic to update the requests pending via *req_update* signal.

The *req_sel[3:0]* signal tells the address generator which buffer is being serviced, it is constructed from the *odd_even_sel* signal and the *color_cnt[2:0]* bus from the interface controller. When data is being transferred to DRAM the word pointer and read pointer for the corresponding buffer are updated. The *req_sel* determines which pointer should be incremented.

```
// determine which buffers need updates
for( i=0; i<12; i++) {
    // determine if request is active, filtered by color enable
    if ( wr_adr[i][3:2] != rd_adr[i][3:2] )
        pend[i] = color_enable[i / 2]
    else
        pend[i] = 0
```

```
                    // determine if any enabled buffer is full
                    if ((wr_adr[i][3:0] - rd_adr[i][3:0]) > 7) AND (color_enable[i / 2] ==.1)) then
                       buf_full = 1
                    )
    // Odd half colors (1,3,5,7,9,11), even half colors (0,2,4,6,8,10)
    odd_pend  = ( pend[1] | pend[3] | pend[5] | pend[7] | pend[9] | pend[11] )
    even_pend = ( pend[0] | pend[2] | pend[4] | pend[6] | pend[8] | pend[10] )
    // fixed servicing order, only update when controller dictates so
    if (req_update == 1) then {
       if (even_pend == 1) then            // even always first
          odd_even_sel = 0
          req_active   = 1
       elsif (odd_pend == 1 ) then         // then check odd
          odd_even_sel = 0
          req_active   = 1
       else                                // nothing active
          odd_even_sel = 0
          req_active   = 0
       )
    // selected requestor
    req_sel[3:0] = (color_cnt[2:0] , odd_even_sel)     // concatentation
```

The read address pointer logic consists of 12 2-bit counters and a word select pointer. The pointers are reset when *dwu_go_pulse* is one. The word pointer (*word_ptr*) is common to all buffers and is used to read out the 64-bit words from the DIU buffer. It is incremented when *buf_rd_en* is active. If the *word_ptr* is 3 and the *buf_rd_en* is active the selected read pointer (*rd_ptr[req_sel]*) will be incremented. A concatenation of the read pointer and the word pointer are use to construct the buffer read address. The read pointers are not reset at the end of each line.

```
    // determine which pointer to update
    if (dwu_go_pulse == 1) then
       rd_ptr[11:0] = 0
       word_ptr     = 0
    elsif (buf_rd_en == 1) then {
       word_ptr++
       if (word_ptr == 3 ) then
          rd_ptr[req_sel]++
       )
    // create the address from the pointer,and word reader
    rd_adr[req_sel] = (rd_ptr[req_sel],word_ptr)     // concatenation
```

The read pointer block determines if the word being read from the DIU buffers is the last word of a line. The buffer address generator indicate the last dot is being written into the buffers via the *line_fin* signal. When received the logic marks the 256-bit word in the buffers as the last word. When the last word is read from the DIU buffer and transferred to DRAM, the flag for that word is reflected to the address generator.

```
    // line end set the flags
    if (dwu_go_pulse == 1) then
       last_flag[1:0][1:0] = 0
    elsif (line_fin == 1 ) then
       // determines the current 256-bit word even been written to
       last_flag[0][wr_adr[0][2]] = 1    // even group flag
       // determines the current 256-bit word odd been written to
       last_flag[1][wr_adr[1][2]] = 1    // odd group flag
    // last word reflection to address generator
    last_wd = last_flag[odd_even_sel][rd_ptr[req_sel][0]]
    // clear the flag
    if (group_fin == 1 ) then
       last_flag[odd_even_sel][rd_ptr[req_sel][0]] = 0
```

When a complete line has been written into the DIU buffers (but has not yet been transferred to DRAM), the buffer address generator block will pulse the *line_fin* signal. The DWU must wait until all enabled

buffers are transferred to DRAM before signaling the LLU that a complete line is available in the dot line store (*dwu_llu_line_wr* signal). When the *line_fin* is received all buffers will require transfer to DRAM. Due to the arbitration, the even group will get serviced first then the odd. As a result the line finish pulse to the LLU is generated from the *last_flag* of the odd group.

```
// must be odd,odd group transfer complete and the last word
dwu_llu_line_wr = odd_even_sel AND group_fin AND last_wd
```

31 Line Loader Unit (LLU)

31.1 OVERVIEW

The Line Loader Unit (LLU) reads dot data from the line buffers in DRAM and structures the data into even and odd dot channels destined for the same print time. The blocks of dot data are transferred to the PHI and then to the printhead. Figure 225 shows a high level data flow diagram of the LLU in context.
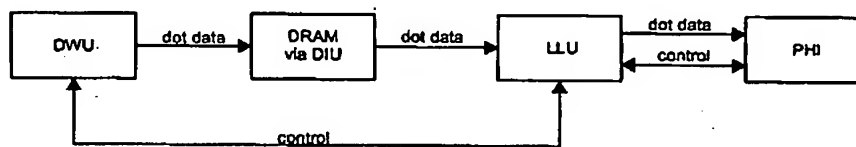

Figure 225. High level data flow diagram of LLU in context

31.2 PHYSICAL REQUIREMENT IMPOSED BY THE PRINTHEAD

The DWU re-orders dot data into 12 separate dot data line FIFOs in the DRAM. Each FIFO corresponds to 6 colors of odd and even data. The LLU reads the dot data line FIFOs and sends the data to the printhead interface. The LLU decides when data should be read from the dot data line FIFOs to correspond with the time that the particular nozzle on the printhead is passing the current line. The interaction of the DWU and LLU with the dot line FIFOs compensates for the physical spread of nozzles firing over several lines at once. For further explanation see Section 30 Dotline Writer Unit (DWU) and Section 32 PrintHead Interface (PHI). Figure 226 shows the physical relationship of nozzle rows and the line time the LLU starts reading from the dot line store.
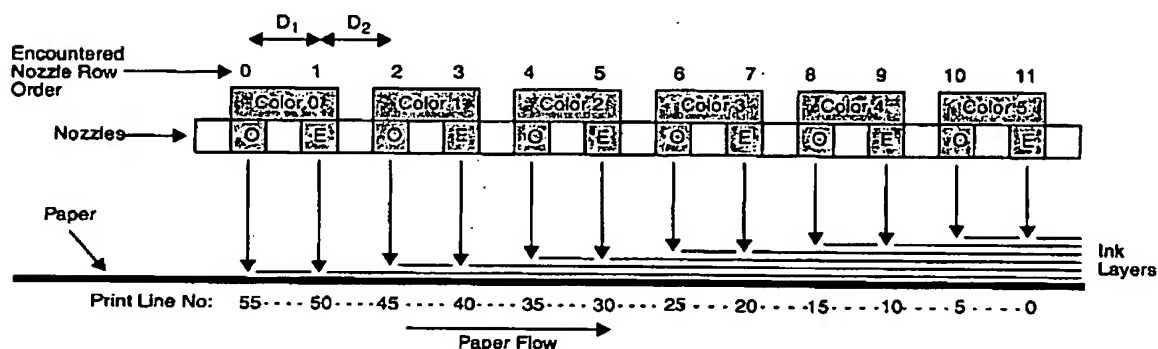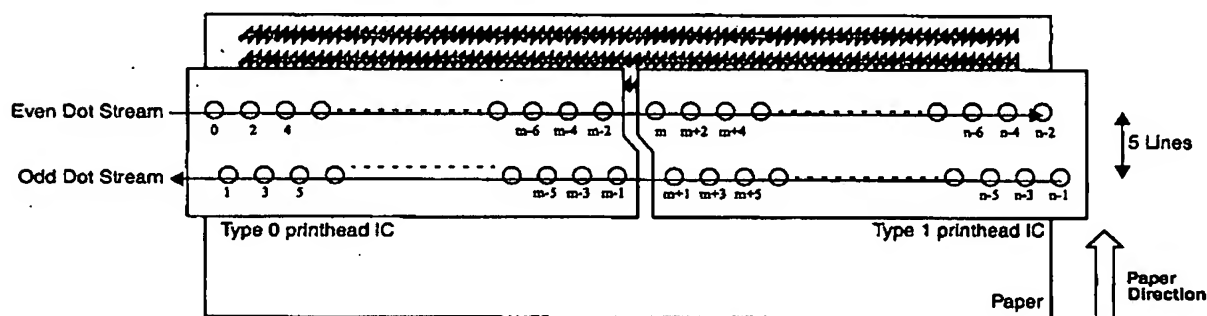

Figure 226. Paper and printhead nozzles relationship (example with $D_1 = D_2 = 5$)

Within each line of dot data the LLU is required to generate an even and odd dot data stream to the PHI block. Figure 227 shows the even and dot streams as they would map to an example bi-lithic printhead. The PHI block determines which stream should be directed to which printhead IC.

M - Midway point in dots
N - Number of dots in a line

Note: Paper passing under printhead

Figure 227. Printhead structure and dot generate order

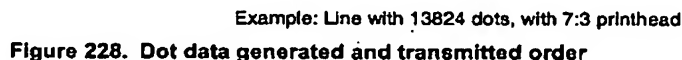## 31.3 DOT GENERATE AND TRANSMIT ORDER

The structure of the printhead ICs dictate the dot transmit order to each printhead IC. The LLU reads data from the dot line FIFO, generates an even and odd dot stream which is then re-ordered (in the PHI) into the transmit order for transfer to the printhead.

The DWU separates dot data into even and odd half lines for each color and stores them in DRAM. It can store odd or even dot data in increasing or decreasing order in DRAM. The order is programmable but for descriptive purposes assume even in increasing order and odd in decreasing order. The dot order structure in DRAM is shown in Figure 219.

The LLU contains 2 dot generator units. Each dot generator reads dot data from DRAM and generates a stream of odd or even dots. The dot order may be increasing or decreasing depending on how the DWU was programmed to write data to DRAM. An example of the even and odd dot data streams to DRAM is shown in Figure 228. In the example the odd dot generator is configured to produce odd dot data in decreasing order and the even dot generator produces dot data in increasing order.

The PHI block accepts the even and odd dot data streams and reconstructs the streams into transmit order to the printhead.

The LLU line size refers to the page width in dots and not necessarily the printhead width. The page width is often the dot margin number of dots less than the printhead width. They can be the same size for full bleed printing.

Generate dot order (to the PHI)

Odd Dot stream | 13823 13821 13819 ... 9749 9747 9745 | 9743 9741 9739 ... 5 3 1 |

Even Dot stream | 0 2 4 ... 9738 9740 9742 | 9744 9746 9748 ... 13818 13820 13822 |

|← 6912 clock cycles →|

Transmit dot order(to the printhead)

Mid
Point

Printhead Channel A | 13823 13821 13819 ... 9749 9747 9745 | 9744 9746 9748 ... 13818 13820 13822 |

Printhead Channel B | 0 2 4 ... 9738 9740 9742 | | 9743 9741 9739 ... 5 3 1 |

|← 4872 clock cycles →|← 2040 clock cycles →|
|← 9744 clock cycles →|

[icon] Even dots from Line Y

[icon] Odd dots from Line Y-5

Example: Line with 13824 dots, with 7:3 printhead

Figure 228. Dot data generated and transmitted order

31.4 LLU START-UP

At the start of a page the LLU must wait for the dot line store in DRAM to fill to a configured level (given by *FifoReadThreshold)* before starting to read dot data. Once the LLU starts processing dot data for a page it must continue until the end of a page, the DWU (and other PEP blocks in the pipeline) must ensure there is always data in the dot line store for the LLU to read, otherwise the LLU will stall, causing the PHI to stall and potentially generate a print error. The *FifoReadThreshold* should be chosen to allow for data rate mismatches between the DWU write side and the LLU read side of the dot line FIFO. The LLU will not generate any dot data until *FifoReadThreshold* level in the dot line FIFO is reached.

Once the *FifoReadThreshold* is reached the LLU begins page processing, the *FifoReadThreshold* is ignored from then on.

When the LLU begins page processing it produces dot data for all colors (although some dot data color may be null data). The LLU compares the line count of the current page, when the line count exceeds the *ColorRelLine* configured value for a particular color the LLU will start reading from that colors FIFO in DRAM. For colors that have not exceeded the *ColorRelLine* value the LLU will generate null data (zero data) and not read from DRAM for that color. *ColorRelLine[N]* specifies the number of lines separating the Nth half color and the first half color to print on that page.

For the example printhead shown in Figure 226, color 0 odd will start at line 0, the remaining colors will all have null data. Color 0 odd will continue with real data until line 5, when color 0 odd and even will contain real data the remaining colors will contain null data. At line 10, color 0 odd and even and color 1 odd will contain real data, with remaining colors containing null data. Every 5 lines a new half color will contain real data and the remaining half colors null data until line 55, when all colors will contain real data. In the example *ColorRelLine[0]* =5, *ColorRelLine[1]* =0, *ColorRelLine[2]* =15, *ColorRelLine[3]* =10 .. etc.

It is possible to turn off any one of the color planes of data (via the *ColorEnable* register), in such cases the LLU will generate zeroed dot data information to the PHI as normal but will not read data from the DRAM.

31.4.1   LLU bandwidth requirements

The LLU is required to generate data for feeding to the printhead interface, the rate required is dependent on the printhead construction and on the line rate configured. The maximum data rate the LLU can produce is 12 bits of dot data per cycle, but the PHI consumes at 12 bits per *phiclk* cycle (2/3 *pclk* rate), i.e. 8 bits per *pclk* cycle. Therefore the DRAM bandwidth requirement for a double buffered LLU is 8 bits per cycle on average. If 1.5 buffering is used then the peak bandwidth requirement is doubled to 16 bits per cycle but the average remains at 8 bits per cycle. Note that while the LLU and PHI could produce data at the 8 bits per cycle rate, the DWU can only produce data at 6 bits per cycle rate.

31.5 IMPLEMENTATION

31.5.1 LLU partition
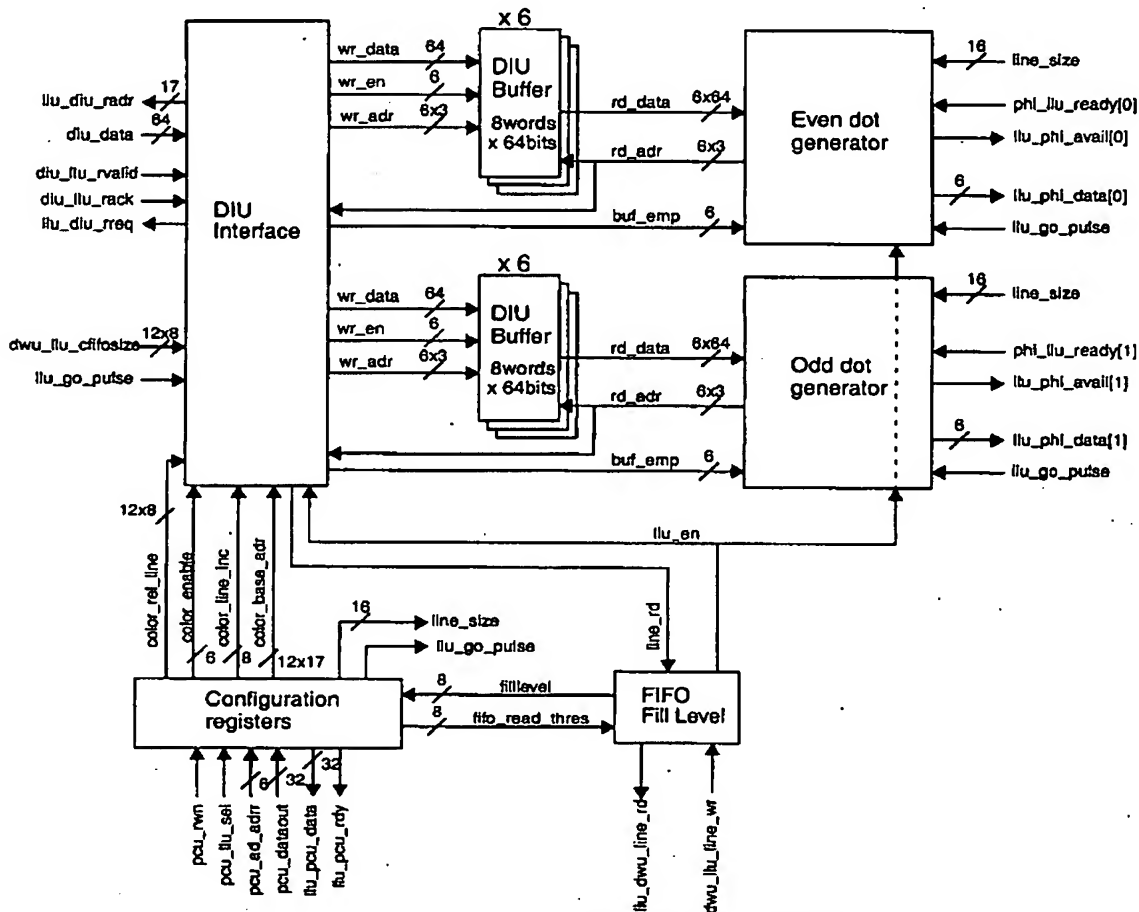
Figure 229. LLU partition

31.5.2 Definitions of I/O

Table 156. LLU I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System clock
prst_n	1	In	System reset, synchronous active low
PHI Interface			

Table 156. LLU I/O definition

Port name	Pins	I/O	Description
llu_phi_data[1:0][5:0]	2x6	Out	Dot Data from LLU to the PHI, each bit is a color plane 5 downto 0. Bus 0 - Even dot data stream Bus 1 - Odd dot data stream Data is active when corresponding bit is active in *llu_phi_avail* bus
phi_llu_ready[1:0]	2	In	Indicates that PHI is ready to accept data from the LLU 0 - Even dot data stream 1 - Odd dot data stream
llu_phi_avail[1:0]	2	Out	Indicates valid data present on corresponding *llu_phi_data*. 0 - Even dot data stream 1 - Odd dot data stream
DIU Interface			
llu_diu_rreq	1	Out	LLU requests DRAM read. A read request must be accompanied by a valid read address.
llu_diu_radr[21:5]	17	Out	Read address to DIU 17 bits wide (256-bit aligned word).
diu_llu_rack	1	In	Acknowledge from DIU that read request has been accepted and new read address can be placed on *llu_diu_radr*
diu_data[63:0]	64	In	Data from DIU to LLU. Each access is 256-bits received over 4 clock cycles First 64-bits is bits 63:0 of 256 bit word Second 64-bits is bits 127:64 of 256 bit word Third 64-bits is bits 191:128 of 256 bit word Fourth 64-bits is bits 255:192 of 256 bit word
diu_llu_rvalid	1	In	Signal from DIU telling LLU that valid read data is on the *diu_data* bus
DWU Interface			
dwu_llu_line_wr	1	In	DWU line write. Indicates that the DWU has completed a full line write. Active high
llu_dwu_line_rd	1	Out	LLU line read. Indicates that the LLU has completed a line read. Active high.
dwu_llu_cfifosize[11:0][7:0]	12x8	In	Indicates the number of lines in the FIFO before the line increment will wrap around in memory.
PCU Interface			
pcu_llu_sel	1	In	Block select from the PCU. When *pcu_llu_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
llu_pcu_rdy	1	Out	Ready signal to the PCU. When *llu_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *llu_pcu_data* is valid.
llu_pcu_data[31:0]	32	Out	Read data bus to the PCU.

31.5.3 Configuration registers

The configuration registers in the LLU are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for a description of the protocol and timing diagrams for reading and writing registers in the

LLU. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the LLU. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *llu_pcu_data*. Table 157 lists the configuration registers in the LLU.

Table 157. LLU registers description

Address LLU_base+	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a LLU block reset.
0x04	Go	1	0x0	Active high bit indicating the LLU is programmed and ready to use. A low to high transition will cause LLU block internal states to reset.
Configuration				
0x08 - 0x38	ColorBaseAdr[11:0]	12x17	0x00000	Specifies the base address (in words) in memory where data from a particular half color (N) will be placed.
0x3C	ColorEnable	6	0x3F	Indicates whether a particular color is active or not. When inactive no data is written to DRAM for that color. 0 - Color off 1 - Color on One bit per color, bit 0 is Color 0 and so on.
0x40	LineSize	16	0x0000	Indicates the number of dots per line.
0x44	FifoReadThreshold	8	0x00	Specifies the number of lines that should be in the FIFO before the LLU starts reading.
0x48 - 0x78	ColorRelLine[11:0]	12x8	0x00	Specifies the relative number of lines to wait from the first before starting to read dot data from the corresponding dot data FIFO Bus 0,1 - Even, Odd line color 0 Bus 2,3 - Even, Odd line color 1 Bus 4,5 - Even, Odd line color 2 Bus 6,7 - Even, Odd line color 3 Bus 8,9 - Even, Odd line color 4 Bus 10,11 - Even, Odd line color 5
Working Registers				
0x7C	FifoFillLevel	8	0x00	Number of lines in the dot line FIFO, line written in but not read out. (Read Only)

A low to high transition of the *Go* register causes the internal states of the LLU to be reset. All configuration registers will remain the same. The block indicates the transition to other blocks via the *llu_go_pulse* signal.

The *ColorLineInc* bus specifies the number of addresses (in 256-bit words) between successive half lines in the dot line store, is used to determine when a half line of data is read from DRAM. It is derived from the *LineSize* register by rounding up the nearest 256-bit value. The same value used for all half colors.

```
if (line_size[7:0] !=0 ) then
   color_line_inc[7:0] = line_size[15:8] + 1
else
   color_line_inc[7:0] = line_size[15:8];
```
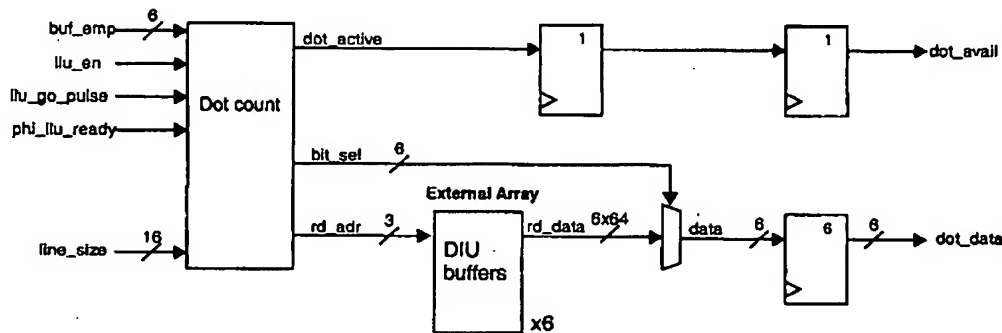
31.5.4   Dot generator

Figure 230. Dot generator RTL Diagram

The dot generator block is responsible for reading dot data from the DIU buffers and sending the dot data in the correct order to the PHI block. The dot generator waits for *llu_en* signal from the fifo fill level block, once active it starts reading data from the 6 DIU buffers and generating dot data for feeding to the PHI.

In the LLU there are two instances of the dot generator, one generating odd data and the other generating even data.

At any time the ready bit from the PHI could be de-asserted, if this happens the dot generator will stop generating data, and wait for the ready bit to be re-asserted.

31.5.4.1 Dot count

In normal operation the dot counter will wait for the *llu_en* and the ready to be active before starting to count. The dot count will produce data as long as the *phi_llu_ready* is active. If the *phi_llu_ready* signal goes low the count will be stalled.

The dot counter increments for each dot that is processed per line. It is used to determine the line finish position, and the bit select value for reading from the DIU buffers. The counter is reset after each line is processed (*line_fin* signal). It determines when a line is finished by comparing the dot count with the configured line size divided by 2 (note that odd numbers of dots will be rounded down).

```
// define the line finish
if (dot_cnt[14:0] == line_size[15:1] )then
   line_fin = 1
else
   line_fin = 0
// determine if word is valid
dot_active = ((llu_en == 1) AND (phi_llu_ready == 1) AND (buf_emp == 0))
// counter logic
if (llu_go_pulse == 1) then
   dot_cnt = 0
elsif ((dot_active == 1)AND (line_fin == 1)) then
   dot_cnt    = 0
elsif (dot_active == 1) then
   dot_cnt    = dot_cnt + 1
else
   dot_cnt    = dot_cnt
// calculate the word select bits
bit_sel[5:0]    := dot_cnt[5:0]
```

The dot generator also maintains a read buffer pointer which is incremented each time a 64-bit word is processed. The pointer is used to address the correct 64-bit dot data word within the DIU buffers. The pointer is reset when *llu_go_pulse* is 1. Unlike the dot counter the read pointer is not reset each line but rounded up the nearest 256-bit word. This allows for more efficient use of the DIU buffers at line finish.

```
// read pointer logic
if (llu_go_pulse == 1) then
   read_adr = 0
elsif (( dot_active == 1) AND (dot_cnt[5:0] = 63 )) then
   read_adr ++                      // normal increment
elsif (( dot_active == 1) AND (line_fin == 1 )) then {
   // special end of line case
   if (dot_cnt[7:0] != 0) then
      read_adr[3:2] ++              // end of line round up
      read_adr[1:0] = 0;
   }
```

31.5.5   Fifo fill level

The LLU keeps a running total of the number of lines in the dot line store FIFO. Every time the DWU signals a line end (*dwu_llu_line_wr* active pulse) it increments the *filllevel*. Conversely if the LLU detects a line end (*line_rd* pulse) the *filllevel* is decremented and the line read is signalled to the DWU via the *llu_dwu_line_rd* signal.

The LLU fill level block is used to determine when the dot line has enough data stored before the LLU should begin to start reading. The LLU at page start is disabled. It waits for the DWU to write lines to the dot line FIFO, and for the fill level to increase. The LLU remains disabled until the fill level has reached the programmed threshold (*fifo_read_thres*). When the threshold is reached it signals the LLU to start processing the page by setting *llu_en* high. Once the LLU has started processing dot data for a page it will not stop if the *filllevel* falls below the threshold.

The line fifo fill level can be read by the CPU via the PCU at any time by accessing the *FifoFillLevel* register. The CPU must toggle the *Go* register in the LLU for the block to be correctly initialized at page start and the fifo level reset to zero.

```
if (llu_go_pulse == 1) then
   filllevel = 0
elsif ((line_rd == 1) AND (dwu_llu_line_wr == 1)) then
   // do nothing
elsif (line_rd == 1) then
   filllevel --
elsif (dwu_llu_line_wr == 1) then
   filllevel ++
// determine the threshold, and set the LLU going
if (llu_go_pulse == 1) then
   llu_en = 0
elsif (filllevel == fifo_read_threshold ) then
   llu_en = 1
```
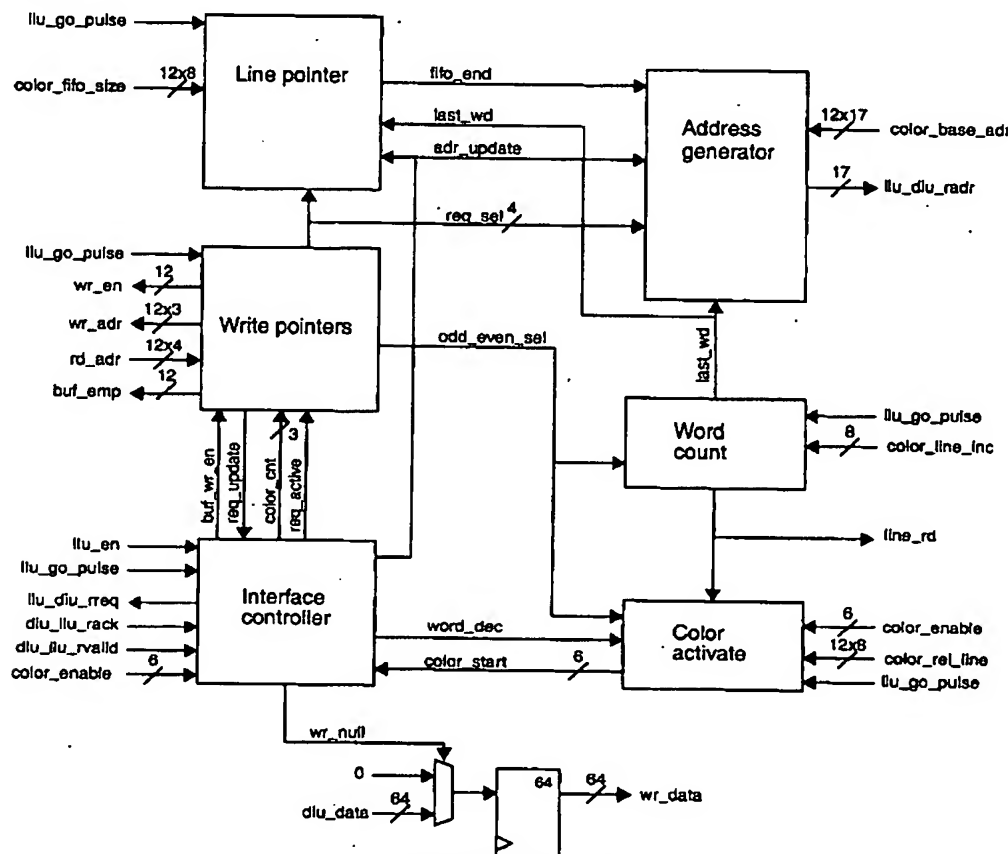
31.5.6 DIU interface

Figure 231. DIU interface

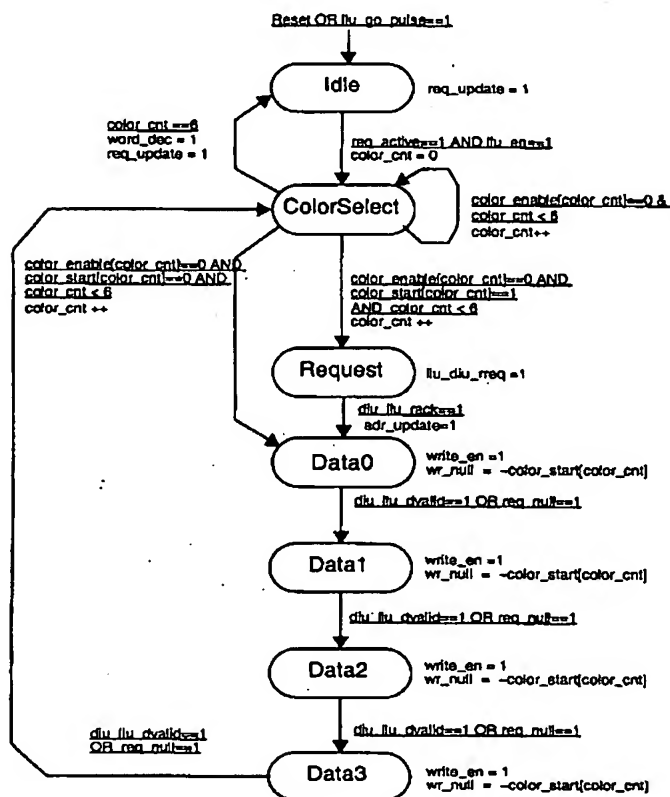#### 31.5.6.1 DIU interface description

The DIU interface block is responsible for determining when dot data needs to be read from DRAM, keeping the dot generators supplied with data and calculating the DRAM read address based on configured parameters, FIFO fill levels and position in a line.

The fill level block enables DIU requests by activating *llu_en* signal. The DIU interface controller then issues requests to the DIU for the LLU buffers to be filled with dot line data (or fill the LLU buffers with null data without requesting DRAM access, if required).

At page start the DIU interface determines which buffers should be filled with null data and which should request DRAM access. New requests are issued until the dot line is completely read from DRAM.

For each request to the DRAM the address generator calculates where in the DRAM the dot data should be read from. The *color_enable* bus determines which colors are enabled, the interface never issues DRAM requests for disabled colors.

31.5.6.2 Interface controller

Figure 232. Interface controller state diagram

The interface controller co-ordinates and issues requests for data transfers from DRAM. The state machine waits in *Idle* state until it is enabled by the LLU controller (*llu_en*) and a request for data transfer is received from the write pointer block.

When an active request is received (*req_active* equals 1) the state machine jumps to the *ColorSelect* state to determine which colors (*color_cnt*) in the group need a data transfer. A group is defined as all odd colors or all even colors. If the color isn't enabled (*color_enable*) the count just increments, and no data is transferred. If the color is enabled, the state machine takes one of two options, either a null data transfer or an actual data transfer from DRAM. A null data transfer writes zero data to the DIU buffer and does not issue a request to DRAM.

The state machine determines if a null transfer is required by checking the *color_start* signal for that color.

If a null transfer is required the state machine doesn't need to issue a request to the DIU and so jumps directly to the data transfer states (*Data0* to *Data3*). The machine clocks through the 4 states each time writing a null 64-bit data word to the buffer. Once complete the state machine returns to the *ColorSelect* state to determine if further transfers are required.

If the *color_start* is active then a data transfer is required. The state machine jumps to the *Request* state and issue a request to the DIU controller for DRAM access by setting *llu_diu_rreq* high. The DIU

responds by acknowledging the request (*diu_llu_rack* equals 1) and then sending 4 64-bit words of data. The transition from *Request* to *Data0* state signals the address generator to update the address pointer (*adr_update*). The state machine clocks through *Data0* to *Data3* states each time writing the 64-bit data into the buffer selected by the *req_sel* bus. Once complete the state machine returns to the *ColorSelect* state to determine if further transfers are required.

When in the *ColorSelect* state and all data transfers for colors in that group have been serviced (i.e. when *color_cnt* is 6) the state machine will return to the *Idle* state. On transition it will update the word counter logic (*word_dec*) and enabled the request logic (*req_update*).

A reset or *llu_go_pulse* set to 1 will cause the state machine to jump directly to *Idle*. The controller will remain in *Idle* state until it is enabled by the LLU controller via the *llu_en* signal. This prevents the DIU attempting the fill the DIU buffers before the dot line store FIFO has filled over its threshold level.

31.5.6.3 Color activate

The color activate logic maintains an absolute line count indicating the line number currently being processed by the LLU. The counter is reset when the *llu_go_pulse* is 1 and incremented each time a *line_rd* pulse is received. The count value (*line_cnt*) is used to determine when to start reading data for a color.

The count is implemented as follows:
```
if ( llu_go_pulse == 1) then
   line_cnt = 0
elsif ( line_rd == 1) then
   line_cnt.++
```

The color activate logic compares line count with the relative line value to determine when the LLU should start reading data from DRAM for a particular half color. It signals the interface controller block which colors are active for this dot line in a page (via the *color_start* bus). It is used by the interface controller to determine which DIU buffers require null data.

Once the *color_start* bit for a color is set it cannot be cleared in the normal page processing process. The bits must be reset by the CPU at the end of a page by transitioning the *Go* bit and causing a pulse on the *llu_go_pulse* signal.

Any color not enabled by the *color_enable* bus will never have its *color_start* bit set.
```
for (i=0; i<12;i++){
   if ( llu_go_pulse == 1) then
      col_on[i] = 0
   elsif ( color_enable[i % 6] == 1 ) then
      col_on[i] = 0
   elsif ( line_cnt == color_rel_line[i]) then
      col_on[i] = 1
   }
// select either odd or even colors
if ( odd_even_sel == 1 ) then  // odd selected
   color_start[5:0] = {col_on[11],col_on[9],col_on[7],col_on[5],col_on[3],col_on[1]}
else                           // even selected
   color_start[5:0] = {col_on[10],col_on[8],col_on[6],col_on[4],col_on[2],col_on[0]}
```

31.5.6.4 Address generator

The address generator block maintains 12 pointers (*color_adr[11:0]*) to DRAM corresponding to current read address in the dot line store for each half color. When a DRAM transfer occurs the address pointer is used first and then updated for the next transfer for the color. The pointer used is selected by the *req_sel* bus, and the pointer update is initiated by the *adr_update* signal from the interface controller.

The pointer update and pointer initialization is dependent on the pointer position in a line and the line position in the FIFO.

When a *llu_go_pulse* is received the pointers are each initialized to the corresponding base address for that color (*color_base_adr*). For each word that is read from DRAM the pointer is incremented. If the word is the last word in a line (*last_wd* equals 1) and the last line in the fifo (*fifo_end* equals 1) then the address pointer is re-initialized to the base address value. The pointer is incremented for all other words.

The address is calculated as follows:

```
// reset to base address
if (llu_go_pulse == 1) then
   color_adr[11:0] = color_base_adr[11:0][21:5]
elsif ( adr_update == 1) then
   if (req_sel == NULL ) then
      // do nothing
   elsif ((fifo_end == 1)AND (last_wd == 1)) then
      color_adr[req_sel]    = color_base_adr[req_sel][21:5]
   else
      color_adr[req_sel] ++   // normal increment
// select the address pointer
llu_diu_radr = color_adr[req_sel]
```

31.5.6.5 Line pointer

The line pointer logic counts the number of dot data lines read from DRAM for each color. The counter value is used to signal the fifo wrap point to the address generator logic. A separate counter is maintained for each color.

The end of a line can be determined when the address is updated (*adr_update* equal 1) and the word transferred is the last word of a line (*last_wd* equal 1). The line pointer that needs to be updated is selected by the *req_sel* bus from the write pointer block. If the selected pointer is zero the counter is reset to the corresponding *color_fifo_size value*, otherwise the counter is decremented.

If the *llu_go_pulse* signal is high the counters are reset to its corresponding *color_fifo_size* value. When the counter is zero it sets the *fifo_end* bit to signal the address generator that the fifo has wrapped (to update the address pointer accordingly).

```
if (llu_go_pulse == 1) then
   line_pt[11:0] = color_fifo_size[11:0]
elsif ((adr_update == 1) AND (last_wd == 1)) then {
   if (line_pt[req_sel] == 0)
      line_pt[req_sel] = color_fifo_size[req_sel]
   else
      line_pt[req_sel] --
   }
// select the correct line pointer for comparison
fifo_end = (line_pt[line_pt] == 0)
```

31.5.6.6 Write pointer

The write pointer logic maintains the buffer write address pointers, determines when the DIU buffers need a data transfer and signals when the DIU buffers are empty. The write pointer determines the address in the DIU buffer that the data should be transferred to.

The write pointer logic compares the read and write pointers of each DIU buffer to determine which buffers require data to be transferred from DRAM (*pend[11:0]* bus), and which buffers are empty (the *buf_emp* signals). Only enabled buffers are considered as indicated by the *color_enable* bus.

Buffers are grouped into odd and even buffers, if an odd buffer requires DRAM access the *odd_pend* signals will be active, if an even buffer requires DRAM access the *even_pend* signals will be active. If both odd and even buffer require DRAM access, the even buffers will get serviced first.

If any buffer requires a DRAM transfer, the logic will indicate to the interface controller via the *req_active* signal, with the *odd_even_sel* signal determining which group of buffers get serviced. The interface controller will check the *color_enable* signal and issue DRAM transfers for all enabled colors in a group. When the transfers are complete it tells the write pointer logic to update the request pending via *req_update* signal.

The *req_sel[3:0]* signal tells the address generator which buffer is being serviced, it is constructed from the *odd_even_sel* signal and the *color_cnt[2:0]* bus from the interface controller. When data is being transferred to DRAM the word pointer and write pointer for the corresponding buffer are updated. The *req_sel* determines which pointer should be incremented.

The write pointer logic operates the same way regardless of whether the transfer is null or not.

```
// determine which buffers need updates
for( i=0; i<12; i++) {
    // determine if request is active, filtered by color enable
    if ( wr_adr[i][3:2] == rd_adr[i][3:2] )
        pend[i] = 1
    else
        pend[i] = 0
    // determine if any enabled buffer is empty
    if ((wr_adr[i][3:0] == rd_adr[i][3:0]) AND (color_enable[i / 2] == 1)) then
        buf_emp[i] = 1
    }
// Odd half colors (1,3,5,7,9,11), even half colors (0,2,4,6,8,10)
odd_pend  = ( pend[1] | pend[3] | pend[5] | pend[7] | pend[9] | pend[11] )
even_pend = ( pend[0] | pend[2] | pend[4] | pend[6] | pend[8] | pend[10] )
// fixed servicing order, only update when controller dictates so
if (req_update == 1) then {
    if (even_pend == 1) then          // even always first
        odd_even_sel = 0
        req_active   = 1
    elsif (odd_pend == 1 ) then       // then check odd
        odd_even_sel = 0
        req_active   = 1
    else                              // nothing active
        odd_even_sel = 0
        req_active   = 0
    }
// selected requestor
req_sel[3:0] = {color_cnt[2:0],odd_even_sel}    // concatentation
```

The write address pointer logic consists of 12 2-bit counters and a word select pointer. The counters are reset when *llu_go_pulse* is one. The word pointer (*word_ptr*) is common to all buffers and is used to write 64-bit words into the DIU buffer. It is incremented when *buf_rd_en* is active. If the *word_ptr* is 3 and the *buf_rd_en* is active the selected write pointer (*wr_ptr[req_sel]*) will be incremented. A concatenation of the write pointer and the word pointer are use to construct the buffer write address. The write pointers are not reset at the end of each line.

```
// determine which pointer to update
if (buf_wr_en == 1) then {
    wr_adr[req_sel]++
    wr_en[req_sel] = 1
    }
```

```
// determine which pointer to update
if (llu_go_pulse == 1) then
   wr_ptr[11:0] = 0
   word_ptr    = 0
elsif (buf_rd_en == 1) then {
   word_ptr++
   if (word_ptr == 3 ) then
      wr_ptr[req_sel]++
   }
// create the address from the write pointer and word pointer
wr_adr[req_sel] = {wr_ptr[req_sel],word_ptr}    // concatenation
```

31.5.6.7 Word count

The word count logic maintains 2 counters to track the number of words transferred from DRAM per line, one counter for odd data, and one counter for even. On receipt of a *llu_go_pulse*, the counters are initialized to the *color_line_inc* value (number of words per line). When a group of words are transferred to DRAM as indicated by the *word_dec* signal from the interface controller, the corresponding counter is decremented. The counter to decrement is indicated by the *odd_even_sel* signal from the write pointer block (even = 0, odd = 1).

When a counter is zero the *last_wd* signal for that group (i.e. odd or even) is set. The *last_wd* signal indicates to the address generator that the next word transferred from DRAM for the corresponding color is the last word in the line. When the last word actually gets transferred the interface controller will pulse the *word_dec* signal causing the corresponding word count to reset to the *color_line_inc* value.

```
// determine which counter to decrement
if (llu_go_pulse == 1) then
   word_cnt[0]  = color_line_inc  // odd count
   word_cnt[1]  = color_line_inc // even count
elsif (word_dec == 1) then {                    // need to decrement one word counter
   if (word_cnt[odd_even_sel] == 0) then        // line finish
      word_cnt[odd_even_sel] = color_line_inc
   else
      word_cnt[odd_even_sel]--
   }
// select the correct the last_wd
last_wd = (word_cnt[odd_even_sel] == 0)
```

The word count logic also determines when a complete line has been read from DRAM, it then signals the fifo fill level logic in both the LLU and DWU (via *line_rd* signal) that a complete line has been read by the LLU (*llu_dwu_line_rd*).

```
// line finish logic
if (llu_go_pulse == 1) then
   line_fin = 0
   line_rd  = 0
elsif ((last_wd == 1) AND (line_fin == 0) AND (word_dec == 1 )) then
   line_fin = 1                      // first group last_wd finish pulse
   line_rd  = 0
elsif ((last_wd == 1) AND (line_fin == 1) AND (word_dec == 1 )) then
   line_fin = 0                      // second group last_wd finish pulse
   line_rd  = 1
else
   line_fin = line_fin              // stay the same
   line_rd  = 0
```

32 PrintHead Interface (PHI)

32.1 OVERVIEW

The Printhead interface (PHI) accepts dot data from the LLU and transmits the dot data to the printhead, using the printhead interface mechanism. The PHI generates the control and timing signals necessary to load and drive the bi-lithic printhead. The CPU determines the line update rate to the printhead and adjusts the line sync frequency to produce the maximum print speed to account for the printhead IC's size ratio and inherent latencies in the syncing system across multiple SoPECs.

The PHI also needs to consider the order in which dot data is loaded in the printhead. This is dependent on the construction of the printhead and the relative sizes of printhead ICs used to create the printhead. See Bi-lithic Printhead Reference document for a complete description of printhead types [10].

The printing process is a real-time process. Once the printing process has started, the next printline's data must be transferred to the printhead before the next line sync pulse is received by the printhead. Otherwise the printing process will terminate with a buffer underrun error.

The PHI can be configured to drive a single printhead IC with or without synchronization to other SoPECs. For example the PHI could drive a single IC printhead (i.e. a printhead constucted with one IC only), or dual IC printhead with one SoPEC device driving each printhead IC.

The PHI interface provides a mechanism for the CPU to directly control the PHI interface pins, allowing the CPU to access the bi-lithic printhead to:
- determine printhead temperature
- test for and determine dead nozzles for each printhead IC
- . initialize each printhead IC
- pre-heat each printhead IC

Figure 233 shows a high level data flow diagram of the PHI in context.
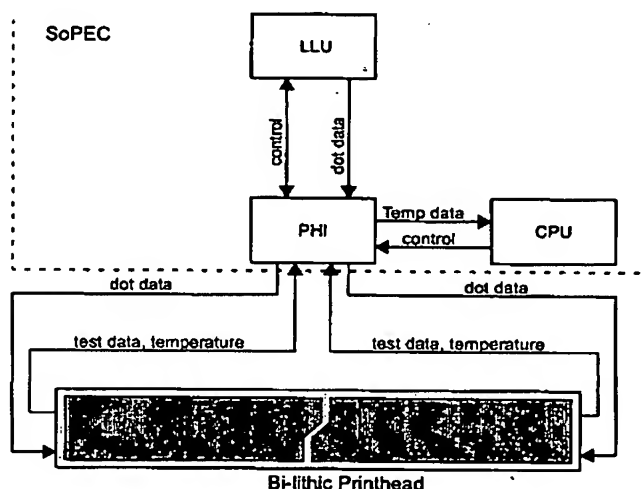

Figure 233. High level data flow diagram of PHI in context

32.2 PRINTHEAD MODES OF OPERATION

The printhead has 4 different modes of operations (although some modes are re-used). The mode of operation is defined by the state of the output pins *phi_lsyncl* and *phi_readl*. As both printhead ICs are driven by the same signals both printhead ICs must be in the same mode of operation. The modes of operation are defined in Table 158.

Table 158. Printhead modes of operation

| Name	phi_readl	phi_lsyncl	State	Description
NORMAL	1	1	N/A	Normal print mode, dot data is clocked into the printhead shift register, on each falling edge of *phi_srclk*
DOT_LOAD/ FIRE_INIT	1	0	*phi_frclk=0*	Dot Load Mode, data stored in the dot shift register is transferred into the dot latch on the falling edge of *phi_lsyncl*, and latched in on the rising edge of *phi_lsyncl*
			phi_srclk=1	Fire load mode. Parameter for generating fire pattern are loaded into generator, data on *phi_ph_data[1:0][0]* is clocked into the generator on each rising edge of *phi_frclk*
TEST_MODE	0	0	*phi_frclk=0*	Dot Load Mode, data stored in the dot shift register is transferred into the dot register on the rising edge of *phi_lsyncl*, identical to DOT_LOAD
			phi_srclk=0	The printhead is in test mode, the temperature delta sigma is clocked out of the printhead on the rising of frclk through *phi_ph_data[1:0][1]* The result of the nozzle test is clocked out of the printhead through *phi_ph_data[1:0][0]*
FIRE_GEN	0	1	N/A	The nozzle test circuit is reset CMOS testing mode, the dot shift register is scanned out of the printhead on the falling edge of *phi_srclk*. Data is output on *phi_ph_data[1:0][1:0]* The initialised generator creates the fire pattern and shift select pattern, and the pattern is clocked into the fire shift register and select shift register on the rising edge of *phi_frclk*

32.3 DATA RATE EQUALIZATION

The LLU can generate dot data at the rate of 12 bits per cycle, where a cycle is at the system clock frequency. In order to achieve the target print rate of 30 sheets per minute, the printhead needs to print a line every 100µs (calculated from 300mm @ 65.2 dots/mm divided by 2 seconds ≈ 100µsec). For a 7:3 constructed printhead this means that 9744 cycles at 106Mhz is quick enough to transfer the dot data. The input FIFOs are used to de-couple the read and write clock domains as well as provide for differences between consume and fill rates of the PHI and LLU.

Nominally the system clock (*pclk*) is run at 160Mhz and the printhead interface clock (*phiclk*) is at 106Mhz.

If the PHI was to transfer data at the full printhead interface rate, the transfer of data to the shorter printhead IC would be completed sooner than the longer printhead IC. While in itself this isn't an issue it requires that the LLU be able to supply data at the maximum rate for short duration, this requires uneven bursty access to DRAM which is undesirable. To smooth the LLU DRAM access requirements over time

the PHI transfers dot data to the printhead at a pre-programmed rate, proportional to the ratio of the shorter to longer printhead ICs.

Without Rate equalization (7:3 head)
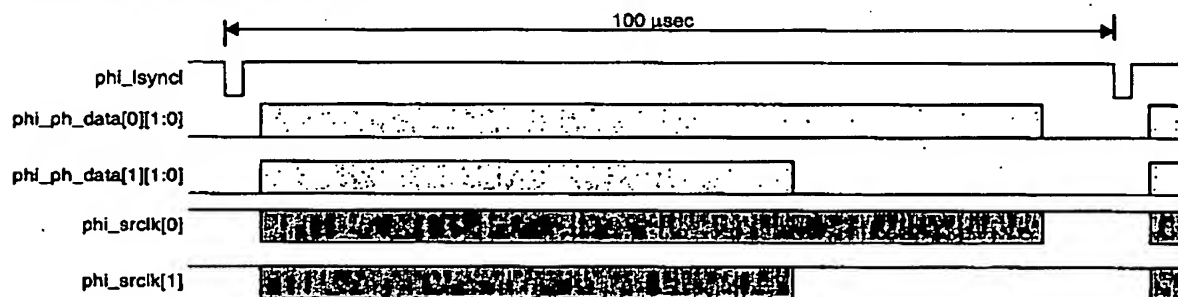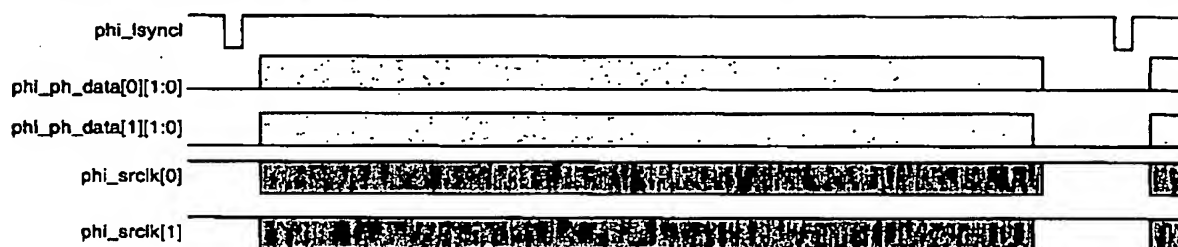

With Rate equalization (7:3 head)

Figure 235. Printhead data rate equalization

The printhead data rate equalization is controlled by *PrintHeadRate[1:0]* registers (one per printhead IC). The register is a 16 bit bitmap of active clock cycles in a 16 clock cycle window. For example if the register is set to 0xFFFF then the output rate to the printhead will be full rate, if it's set to 0xF0F0 then the output rate is 50% where there is 4 active cycles followed by 4 inactive cycles and so on. If the register was set to 0x0000 the rate would be 0%. The relative data transfer rate of the printhead can be varied from 0-100% with a granularity of 1/16 steps.

Table 159. Example rate equalization values for common printheads

Printhead Ratio A-B	Printhead A rate (%)	Printhead B rate (%)
8:2	0xFFFF (100%)	0x1111 (25%)
7:3	0xFFFF (100%)	0x5551 (43.7%)
6:4	0xFFFF (100%)	0xF1F2 (68.7%)
5:5	0xFFFF (100%)	0xFFFF (100%)

If both printhead ICs are the same size (e.g. a 5:5 printhead) it may be desirable to reduce the data rate to both printhead ICs, to reduce the read bandwidth from the DRAM.

32.4 DOT GENERATE AND TRANSMIT ORDER

Several printhead types and arrangements exists (see Section 35 Memjet Printhead) . The PHI is capable of driving all possible configurations, but for the purposes of simplicity only one arrangement (arrangement 0 - see Section 35 Memjet Printhead) is described in the following examples.
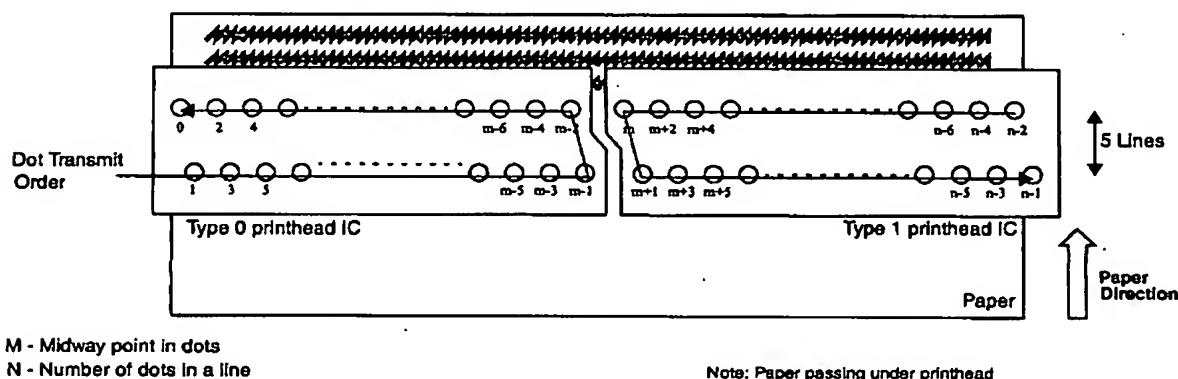

M - Midway point in dots
N - Number of dots in a line

Note: Paper passing under printhead

Figure 236. Printhead structure and dot generate order

The structure of the printhead ICs dictate the dot transmit order to each printhead IC. The PHI accepts two streams of dot data from the LLU, one even stream the other odd. The PHI constructs the dot transmit order streams from the dot generate order received from the LLU. Each stream of data has already been arranged in increasing or decreasing dot order sense by the DWU. The exact sense choice is dependent on the type of printhead ICs used to construct the printhead, but regardless of configuration the odd and even stream should be of opposing sense.

The dot transmit order is shown in Figure 236. Dot data is shifted into the printhead in the direction of the arrow, so from the diagram (taking the type 0 printhead IC) even dot data is transferred in increasing order to the mid point first (0, 2, 4, ..., m-6, m-4, m-2), then odd dot data in decreasing order is transferred (m-1, m-3, m-5,...., 5, 3, 1). For the type 1 printhead IC the order is reversed, with odd dots in increasing order transmitted first, followed by even dot data in decreasing order. Note for any given color the odd and even dot data transferred to the printhead ICs are from different dot lines, in the example in the diagram they are separated by 5 dot lines. Table 160 shows the transmit dot order for some common A4 printheads. Different type printheads may have the sense reversed and may have an odd before even transmit order or vice versa.

Table 160. Example printhead ICs, and dot data transmit order for A4 (13824 dots) page

Size	Dots	Dot Order	
Type 0 Printhead IC			
8	11160	0,2,4,8.....,5574,5576,5578	5579,5577,5575......7,5,3,1
7	9744	0,2,4,8.....,4866,4868,4870	4871,4869,4867.....7,5,3,1
6	8328	0,2,4,8.....,4158,4160,4162	4163,4161,4159......7,5,3,1
5	6912	0,2,4,8.....,3450,3452,3454	3455,3453,3451......7,5,3,1
4	5496	0,2,4,8.....,2742,2744,2746	2847,2845,2843......7,5,3,1
3	4080	0,2,4,8.....,2034,2036,2038	2039,2037,2035......7,5,3,1

Table 160. Example printhead ICs, and dot data transmit order for A4 (13824 dots) page

Site	# Dots	Dot order	
2	2664	0,2,4,8......,1326,1328,1330	1331,1329,1327......7,5,3,1
Type 1 Printhead IC			
8	11160	13823,13821,13819 ......,1337,1335,1333	1332,1334,1336.......13818,13820,13822
7	9744	13823,13821,13819 ......,2045,2043,2041	2040,2042,2044.......13818,13820,13822
6	8328	13823,13821,13819 ......,2853,2851,2849	2848,2850,2852.......13818,13820,13822
5	6912	13823,13821,13819 ......,3461,3459,3457	3456,3458,3460.......13818,13820,13822
4	5496	13823,13821,13819 ......,4169,4167,4165	4164,4166,4168.......13818,13820,13822
3	4080	13823,13821,13819 ......,4877,4875,4873	4872,4874,4876.......13818,13820,13822
2	2664	13823,13821,13819 ......,5585,5583,5581	5580,5582,5584.......13818,13820,13822

32.4.1 Dual Printhead IC

Generate dot order (from the LLU)

Figure 237. Dot data generated and transmitted order

The LLU contains 2 dot generator units. Each dot generator reads dot data from DRAM and generates a stream of dots in increasing or decreasing order. A dot generator can be configured to produce odd or even dot data streams, and the dot sense is also configurable. In Figure 237 the odd dot generator is configured to produce odd dot data in decreasing order and the even dot generator produces dot data in increasing order.

In order to reconstruct the dot data streams from the generate order to the transmit order, the connection between the generators and transmitters needs to be switched at the mid point. At line start the odd dot generator feeds the type 1 printhead, and the even dot generator feeds the type 0 printhead. This continues until both printheads have received half the number of dots they require (defined as the mid point). The mid point is calculated from the configured printhead size registers (*PrintHeadSize*). Once both printheads have reached the mid point, the PHI switches the connections between the dot generators and the printhead, so now the odd dot generator feeds the type 0 printhead and the even dot generator feeds the type 1 printhead. This continues until the end of the line.

It is possible that both printheads will not be the same size and as a result one dot generator may reach the mid point before the other. In such cases the quicker dot generator is stalled until both dot generators reach the mid point, the connections are switched and both dot generators are restarted.

Note that in the example shown in Figure 237 the dot generators could generate an A4 line of data in 6912 cycles, but because of the mismatch in the printhead IC sizes the transmit time takes 9744 cycles.

32.4.2 Single printhead IC

In some cases only one printhead IC may be connected to the PHI. In Figure 238 the dot generate and transmit order is shown for a single IC printhead of 9744 dots width. While the example shows the print-head IC connected to channel A, either channel could be used. The LLU generates odd and even dot streams as normal, it has no knowledge of the physical printhead configuration. The PHI is configured with the printhead size (*PrintHeadSize[1]* register) for channel B set to zero and channel A is set to 9744.

Generate dot order (from the LLU)
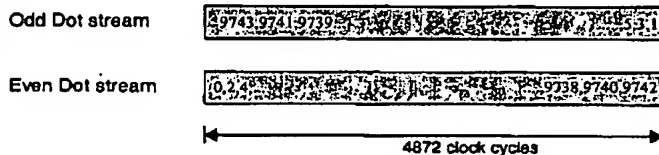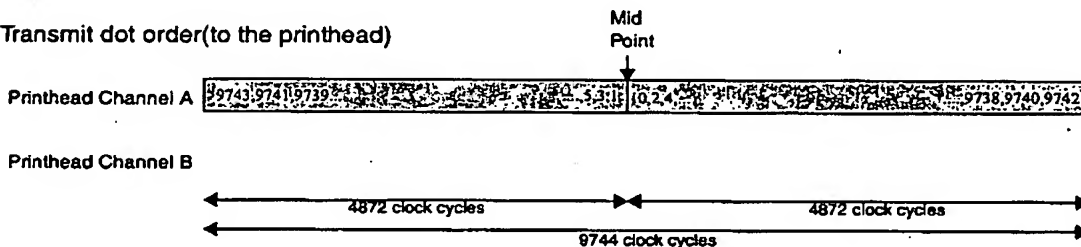
Figure 238. Dot data generated and transmitted order (single printhead case)

Note that in the example shown in Figure 238 the dot generators could generate an 7 inch line of data in 4872 cycles, but because the printhead is using one IC, the transmit time takes 9744 cycles, the same speed as an A4 line with a 7:3 printhead.

32.4.3 Summary of generate and transmit order requirements

In order to support all the possible printhead arrangements, the PHI (in conjuction with the LLU/DWU) must be capable of re-ordering the bits according to the following criteria:
- Be able to output the even or odd plane first.
- Be able to output even and odd planes independently.
- Be able to reverse the sequence in which the color planes of a single dot are output to the printhead.

32.5   PRINT SEQUENCE

The PHI is responsible for accepting dot data streams from the LLU, restructuring the dot data sequence and transferring the dot data to each printhead within a line time (i.e before the next line sync).

Before a page can be printed the printhead ICs must be initialized. The exact initialization sequence is configuration dependent, but will involve the fire pattern generation initialization and other optional steps. The initialization sequence is implemented in software.

Once the first line of data has been transferred to the printhead, the PHI will interrupt the CPU by asserting the *phi_icu_print_rdy* signal. The interrupt can be optionally masked in the ICU and the CPU can poll the signal via the PCU or the ICU. The CPU must wait for a print ready signal in all printing SoPECs before starting printing.

Once the CPU in the PrintMaster SoPEC is satisfied that printing should start, it triggers the LineSync-Master SoPEC by writing to the *PrintStart* register of all printing SoPECs. The transition of the *PrintStart* register in the LineSyncMaster SoPEC will trigger the start of *lsyncl* pulse generation. The PrintMaster and LineSyncMaster SoPEC are not necessarily the same device, but often are the same. For a more in depth definition see section 12.3 Multi-SoPEC systems on page 104.

Writing to the *PrintStart* register generates a pulse which is used to generate the line sync in the LineSyncMaster which is in turn used to align all SoPECs in a multi-SoPEC system. All printhead signaling is aligned to the line sync. The *PrintStart* is only used to align the first line sync in a page.

When a SoPEC receives a line sync pulse it means that the line previously transferred to the printhead is now printing, so the PHI can begin to transfer the next line of data to the printhead. When the transfer is complete the PHI will wait for the next line sync pulse before repeating the cycle. If a line sync arrives before a complete line is transferred to the printhead (i.e. a buffer error) the PHI generates a buffer underrun interrupt, and halts the block.

For each line in a page the PHI must transfer a full line of data to the printhead before the next line sync is generated or received.

32.5.1   Sync pulse control

If the PHI is configured as the LineSyncMaster SoPEC it will start generating line sync signals *LsyncPre* number of *phiclk* cycles after *PrintStart* register rising transition is detected. All other signals in the PHI interface are referenced from the falling edge of *phi_lsyncl* signal.

If the SoPEC is in line sync slave mode it will receive a line sync pulse from the LineSyncMaster SoPEC through the *phi_lsyncl* pin which will be programmed into input mode. The *phi_lsyncl* input pin is treated as an asynchronous input and is passed through a de-glitch circuit of programmable de-glitch duration (*LsyncDeglitchCnt*).

The *phi_lsyncl* will remain low for *LsyncLow* cycles, and then high for *LsyncHigh* cycles. The *phi_lsyncl* profile is repeated until the page is complete. The period of the *phi_lsyncl* is given by *LsyncLow* + *LsyncHigh* cycles. Note that the *LsyncPre* value is only used to vary the time between the generation of the first *phi_lsyncl* and the *PageStart* indication from the CPU. See Figure 239 for reference diagram.

If the SoPEC device is in line sync slave mode, the *LsyncMinPeriod* register specifies the minimum allowed *phi_lsyncl* period. Any *phi_lsyncl* pulses received before the *LsyncMinPeriod* has expired will trigger a buffer underrun error.

32.5.2   Shift register signal control

Once the PHI receives the line sync pulse, the sequence of data transfer to the printhead begins. All PHI control signals are specified from the falling edge of the line sync.

The *phi_srclk* (and consequently *phi_ph_data*) is controlled by the *SrclkPre, SrclkPost* registers. The *SrclkPre* specifies the number of *phiclk* cycles to wait before beginning to transfer data to the printhead. Once data transfer has started, the profile of the *phi_srclk* is controlled by *PrintHeadRate* register and the status of the PHI input FIFO. For example it is possible that the input FIFO could empty and no data would be transferred to the printhead while the PHI was waiting. After all the data for a printhead is transferred to the PHI, it counts *SrclkPost* number of *phiclk* cycles. If a new *phi_lsyncl* falling edge arrives before the count is complete the PHI will generate a buffer underrun interrupt (*phi_icu_underrun*).

32.5.3 Firing sequence signal control

The profile of the *phi_frclk* pulses per line is determined by 4 registers *FrclkPre, FrclkLow, FrclkHigh, FrclkNum*. The *FrclkPre* register specifies the number of cycles between line sync falling edge and the *phi_frclk* pulse high. It remains high for *FrclkHigh* cycles and then low for *FrclkLow* cycles. The number of pulses generated per line is determined by *FrclkNum* register.

The *phi_profile* pin is specified in a similar manner by the *ProfilePre, ProfileLow, ProfileHigh, ProfileNum* registers.

The *phi_frclk* period and the *phi_profile* period should be programmed the same, so *FrclkHigh + FrclkLow* should equal the *ProfileHigh + ProfileLow*, and the number of cycles for each in a line time should also be equal i.e. *FrclkNum = ProfileNum*.

The total number of cycles required to complete a firing sequence should be less than the *phi_lsyncl* period i.e. ((*ProfileHigh + ProfileLow*) * *ProfileNum*) + *ProfilePre* < (*LsyncLow + LsyncHigh*).
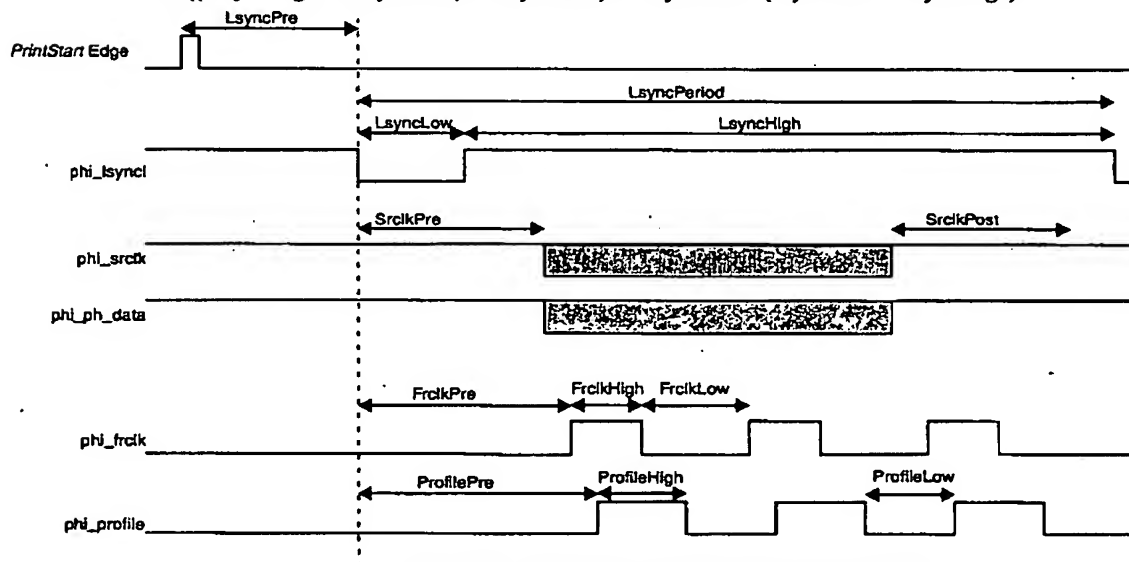

Figure 239. Printhead interface timing parameters

Figure 239 details the timing parameters controlling the PHI. All timing parameters are measured in number of *phiclk* cycles.

32.5.4 Page complete

The PHI counts the number of lines processed through the interface. The line count is initialised to the *PageLenLine* and decrements each time a line is processed. When the line count is zero it pulses the

phi_icu_page_finish signal. A pulse on the *phi_icu_page_finish* automatically resets the PHI *Go* register, and can optionally cause an interrupt to the CPU. Should the page terminate abnormally, i.e. a buffer underrun, the *Go* register will be reset and an interrupt generated.

32.5.5   Line sync interrupt

The PHI will generate an interrupt to the CPU after a predefined number of line syncs have occured. The number of line syncs to count is configured by the *LineSyncInterrupt* register. The interrupt can be disabled by setting the register to zero.

32.6   DOT LINE MARGIN

The PHI block allows the generation of margins either side of the received page from the LLU block. This allows the page width used within PEP blocks to differ from the physical printhead size.

This allows SoPEC to store data for a page minus the margins, resulting in less storage requirements in the shared DRAM and reduced memory bandwidth requirements. The difference between the dot data line size and the line length generated by the PHI is the dot line margin length. There are two margins specified for any sheet, a margin per printhead IC side.

The margin value is set by programming the *DotMargin* register per printhead IC. It should be noted that the *DotMargin* register represents half the width of the actual margin (either left or right margin depending on paper flow direction). For example, if the margin in dots is 1 inch (1600 dots), then *DotMargin* should be set to 800. The reason for this is that the PHI only supports margin creation cases 1 and 3 described below.
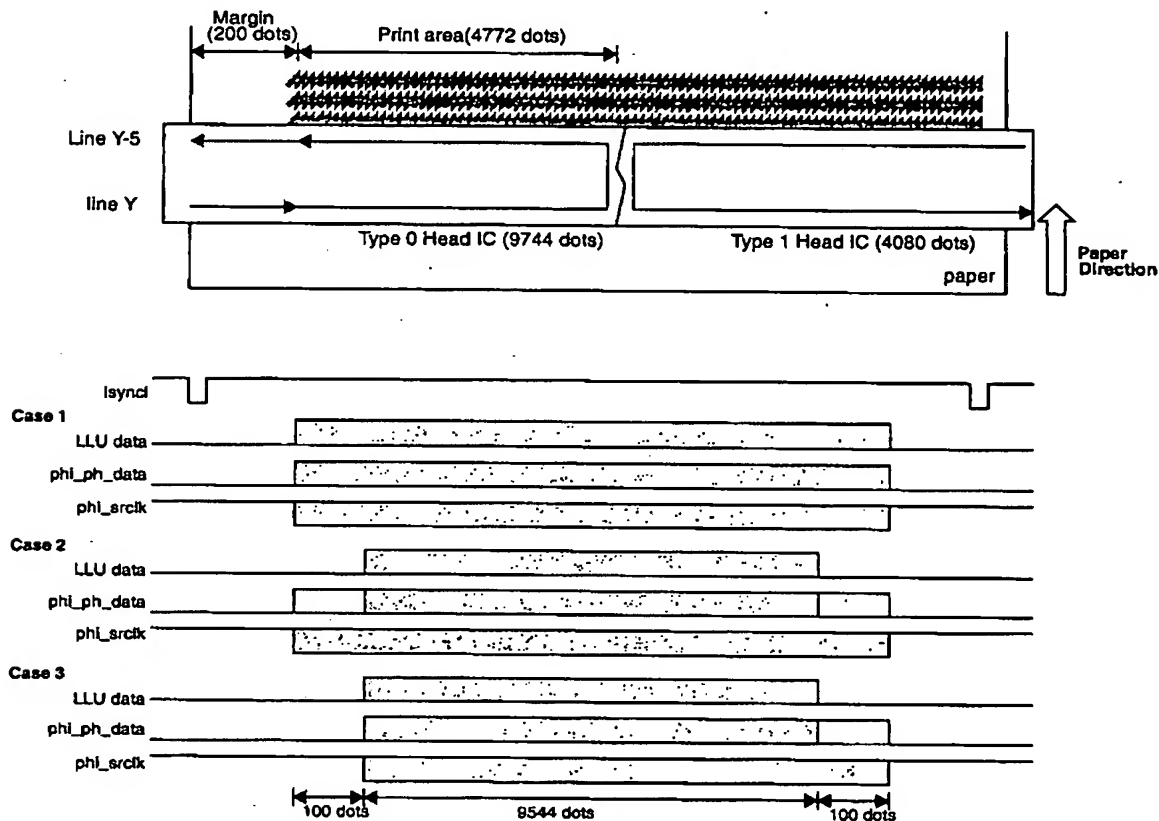
See example in Figure 240.

Figure 240. Printhead timing with margining

In the example the margin for the type 0 printhead IC is set at 100 dots (*DotMargin*=100), implying an actual margin of 200 dots.

If case one is used the PHI takes a total of 9744 *phi_srclk* cycles to load the dot data into the type 0 printhead. It also requires 9744 dots of data from the LLU which in turn gets read from the DRAM. In this case the first 100 and last 100 dots would be zero but are processed though the SoPEC system consuming memory and DRAM bandwidth at each step.

In case 2 the LLU no longer generates the margin dots, the PHI generates the zeroed out dots for the margining. The *phi_srclk* still needs to toggle 9744 times per line, although the LLU only needs to generate 9544 dots giving the reduction in DRAM storage and associated bandwidth. The case 2 senario is not supported by the PHI because the same effect can be supported by means of case 1 and case 3.

If case 3 is used the benefits of case 2 are achieved, but the *phi_srclk* no longer needs to toggle the full 9744 clock cycles. The *phi_srclk* cycles count can be reduced by the margin amount (in this case 9744-100=9644 dots), and due to the reduction in *phi_srclk* cycles the *phi_lsyncl* period could also be reduced, increasing the line processing rate and consequently increasing print speed. Case 3 works by shifting the

odd (or even) dots of a margin from line Y to become the even (or odd) dots of the margin Y-4, (Y-5 adjusted due to being printed one line later). This works for all lines with the exception of the first line where there has been no previous line to generate the zeroed out margin. This situation is handled by adding the line reset sequence to the printhead initialization procedure, and is repeated between pages of a document. See section 32.8.3 on page 512.

32.7 DOT COUNTER

For each color the PHI keeps a dot usage count for each of the color planes (called *AccumDotCount*). If a dot is used in particular color plane the corresponding counter is incremented. Each counter is 32 bits wide and saturates if not reset. A write to the *DotCountSnap* register causes the *AccumDotCount[N]* values to be transferred to the *DotCount[N]* registers (where N is 5 to 0, one per color). The *AccumDotCount* registers are cleared on value transfer.

The *DotCount[N]* registers can be written to or read from by the CPU at any time. On reset the counters are reset to zero.

The dot counter only count dots that are passed from the LLU through ther PHI to the printhead. Any dots generated by direct CPU control of the PHI pins will not be counted.

32.8 CPU IO CONTROL

The PHI interface provides a mechanism for the CPU to directly control the PHI interface pins, allowing the CPU to access the bi-lithic printhead:
- Determine printhead temperature
- Test for and determine dead nozzles for each printhead IC
- Printhead IC initialization
- Printhead pre-heat function

The CPU can gain direct control of the printhead interface connections by setting the *PrintHeadCpuCtrl* register to one. Once enabled the printhead bits are driven directly by the *PrintHeadCpuOut* control register, where the values in the register are reflected directly on the printhead pins and the status of the printhead input pins can be read directly from the *PrintHeadCpuIn*. The direction of pins is controlled by programming *PrintHeadCpuDir* register. The register to pin mapping is as follows:

Table 161. CPU control and status registers mapping to printhead interface

Register Name	bits	Printhead pin
PrintHeadCpuOut	1:0	phi_ph_data_o[0][1:0]
	3:2	phi_ph_data_o[1][1:0]
	4	phi_lsync_o
	5	phi_readl
	7:6	phi_srclk[1:0]
	8	phi_frclk
	9	phi_profile

Table 161. CPU control and status registers mapping to printhead interface

Register Name	bits	Printhead pin
PrintHeadCpuDir	1:0	phi_ph_data_e[0][1:0] direction control. 1 - output mode 0 - input mode
	3:2	phi_ph_data_e[1][1:0] direction control 1 - output mode 0 - input mode
	4	phi_lsyncl_e direction control 1 - output mode 0 - input mode
PrintHeadCpuIn	1:0	phi_ph_data_i[0][1:0]
	3:2	phi_ph_data_i[1][1:0]
	4	phi_lsyncl_i

It is important to note that once in *PrintHeadCpuCtrl* mode it is the responsibility of the CPU to drive the printhead correctly and not create situations where the printhead could be destroyed such as activating all nozzles together.

Note the following procedures are based on current printhead capabilities, and are subject to change.

32.8.1 Dead nozzle information capture

The CPU (via the direct printhead control mechanism) has the capability of testing each of the nozzles in the printhead and determining which nozzles are dead, the resultant dead nozzle information is processed by the CPU to generate the dead nozzle table used by the DNC.

32.8.1.1 Nozzle test procedure

The nozzle test software must first initialize the fire pattern generator for each printhead IC as normal, then it must initialize the fire pattern register as normal. The fire pattern generator parameters must be chosen so as to create a fire pattern where only one nozzle is firing at a time.

For example if the printhead is constructed with a 7:3 configuration where the left printhead is 7 inches and the right 3 inches. The fire pattern length is equal to the number of dots in a half line (NLEN=n-1,where n = 9744 / 2 = 4872), the COUNT=1 and B=0. The fire generator in the printhead needs to be initialized with NLEN=4871, COUNT=1, B=0. See Section 32.8.4 for exact details on how to program the fire pattern generator.

Once the generator is setup the nozzle test software puts the printhead into FIRE_GEN mode and the fire pattern is loaded into the fire shift registers.

The next step is to load the dot data shift registers with a test pattern. Any test pattern could be used it should be chosen so as to allow only one color to fire at a time. Once the printhead shift registers are initialized the software can begin the nozzle test sequence.

The printhead is put in FIRE_GEN mode which resets the test circuit, both *phi_srclk* and *phi_frclk* are held inactive. After a pre-determined time the printhead is put in TEST_MODE where the nozzle is tested.

The test software toggles *phi_profile* output pin and then samples the test result on the *phi_ph_data* pin. The test software then generates one *phi_frclk* pulse to advance the fire pattern and repeats the profile pulse and test result capture as before. This procedure is repeated for all dots in the half dot line. Once the test result for a particular dot line is complete the whole procedure is repeated 12 times once for each half dot line.

The dead nozzle software collates all the nozzles test results and produces the dead nozzle table for use by the DNC.
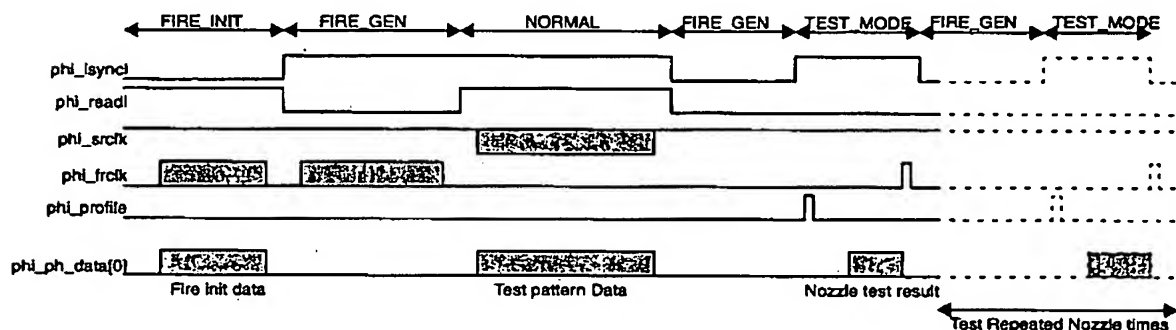

Figure 241. Nozzle Test Modes & Setup

32.8.2 Temperature capture

Occasionally the CPU will need to sample the printhead temperature and possibly adjust the firing profile based on the result.

To capture the printhead temperature, the printhead must be put into TEST_MODE, and the *phi_ph_data_i* pin input mode. The CPU will toggle the *phi_frclk* and then sample the *phi_ph_data_i* to capture the temperature data. The cycle is repeated N times, and the N bits of data are used to generate the printhead temperature value. The temperature capture waveform is shown in Figure 242.

The exact number of bits required (i.e. N) and the temperature value generation mechanism is currently undefined.
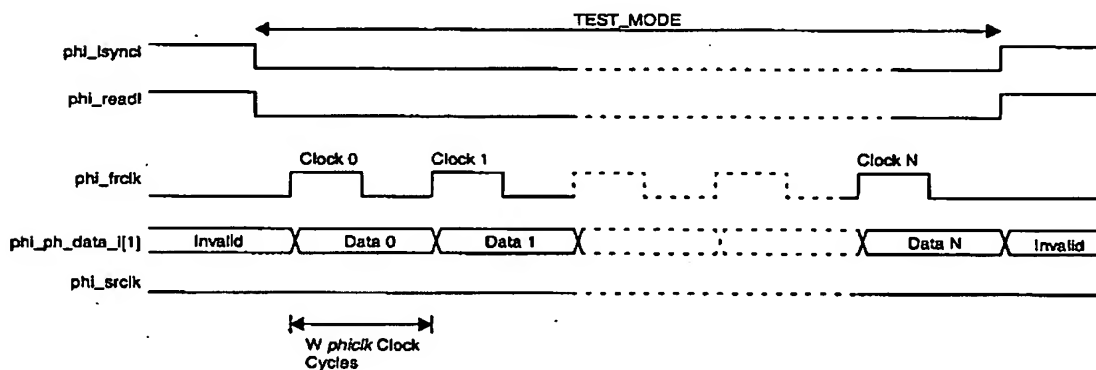

Figure 242. Temperature Capture Waveform

32.8.3 Printhead initialization procedure

In order to use the printhead for the first time the CPU must download parameters for controlling the fire pattern generator. The download is performed by entering the FIRE_INIT mode and data is transferred through the *phi_ph_data[1:0][0]* pins (one pin per printhead IC) and clocked into the printhead on the rising edge of *phi_frclk*. In total 29 clock cycles are required to transfer the full set of parameters.

Table 162. Parameters for Fire Pattern Initialization

Name	Bits	Description
NLEN	14	Fire pattern length. Values defines the length of the fire pattern. NLEN=N-1 where N is the pattern length.
COUNT	14	Defines the remaining number of clock cycles required to generate the Fire Pattern. Is given by COUNT= $(L_a/2)$ Mod $N-1$ where L_a is the dot length of longer printhead or COUNT= $(L_a - L_b -((L_b/2) \bmod N))$ Mod $N-1$ for the shorter printhead
B	1	Select shift register inversion bit.

Once the generator is initialized the fire pattern and select pattern need to be created and shifted into their respective shift registers. The printheads are put into FIRE_GEN mode and the *phi_frclk* is toggled L_a times, where L_a is the length of the longer printhead in dots. As *phi_frclk* is a common signal for both printheads it means that if the printhead ICs are of different length one printhead IC will get clocked too many times by *phi_frclk*. The fire pattern generator internal in each printhead IC takes account of this. See Section 32.8.4 Fire pattern generator.

If dot line margining is to be used the dot data registers in the margining region in the printhead IC need to be initialized to zero before any line is printed. See section 32.6 on page 507 for a full explanation of dot line margin setup. The CPU does this by entering NORMAL_MODE and fills the dot data shift register with zeros. This is performed by clocking the *phi_srclk* to each printhead dot margin times for the each printhead IC. As *phi_srclk* is not common to both printhead ICs the number of clock cycles can be differed to each printhead IC.

Once the printhead initialization is complete control of the printhead can be released to the PHI to allow printing to begin.

32.8.4 Fire pattern generator

The fire pattern generator is logic within each printhead IC used to generate the fire pattern and the select shift pattern. The fire pattern generator must be initialized by the SoPEC device before a page can be printed. The SoPEC uses the CPU direct IO control of the printhead pins to download the initialization parameters and generate the initialization sequence.

32.9   IMPLEMENTATION

32.9.1   Definitions of I/O

Table 163. Printhead Interface I/O definition

Port name	Pins	I/O	Description
Clocks and Resets			
pclk	1	In	System Clock
phiclk	1	In	Printhead interface clock (doclk/3) used to transfer data from pclk to doclk domains
doclk	1	In	Data out clock (2x pclk) used to transfer data to printhead
prst_n	1	In	System reset, synchronous active low. Synchronous to pclk
phirst_n	1	In	System reset, synchronous active low. Synchronous to phiclk
dorst_n	1	In	System reset, synchronous active low. Synchronous to doclk
General			
phi_icu_print_rdy	1	Out	Indicates that the first line of data is transferred to the printhead Active high.
phi_icu_page_finish	1	Out	Indicates that data for a complete page has transferred. Active high
phi_icu_underrun	1	Out	Indicates the PHI has detected a buffer underrun. Active high
phi_icu_linesync_int	1	Out	Indicates the PHI has detected LineSyncInterrupt number of line syncs.
Debug			
debug_data_out[2:0]	3	In	Output debug data to be muxed on to the PHI pins
debug_cntrl[2:0]	3	In	Control signal for each PHI bound debug data line indicating whether or not the debug data should be selected by the pin mux
LLU Interface			
llu_phi_data[1:0][5:0]	2x6	Out	Dot Data from LLU to the PHI, each bit is a color plane 5 downto 0. Bus 0 - Even dot data stream Bus 1 - Odd dot data stream Data is active when corresponding bit is active in llu_phi_avail bus
phi_llu_ready[1:0]	2	In	Indicates that PHI is ready to accept data from the LLU 0 - Even dot data stream 1 - Odd dot data stream
llu_phi_avail[1:0]	2	Out	Indicates valid data present on corresponding llu_phi_data. 0 - Even dot data stream 1 - Odd dot data stream
Printhead Interface			
phi_ph_data_i[1:0][1:0]	2x2	In	Dot data input from printhead. Bus 0 - Printhead channel A Bus 1 - Printhead channel B
phi_ph_data_o[1:0][1:0]	2x2	Out	Dot data output to printhead. Each bus to each printhead contains 2 bits of data Bus 0 - Printhead channel A Bus 1 - Printhead channel B
phi_ph_data_e[1:0][1:0]	2x2	Out	Dot data direction control. Pin is driving when high Bus 0 - Printhead channel A Bus 1 - Printhead channel B
phi_srclk[1:0]	2	Out	Dot data shift clock used to clock in printhead data Bus 0 - Printhead channel A Bus 1 - Printhead channel B

Table 163. Printhead Interface I/O definition

| Port name	Pins	I/O	Description
phi_readl	1	Out	Common printhead mode control. Used in conjunction with *phi_lsyncl* to determine the printhead mode 0 - SoPEC receiving, printhead driving 1 - SoPEC driving, printhead receiving
phi_frclk	1	Out	Common Fire pattern clock needs to toggle once per fire cycle
phi_profile	1	Out	Common pulse profile for all colors
phi_lsyncl_o	1	Out	Capture dot data for next print line, output mode
phi_lsyncl_e	1	In	*phi_lsyncl* output enable, when high *phi_lsyncl* pin is driving
phi_lsyncl_i	1	In	Line Sync Pulse from Master SoPEC
PCU Interface			
pcu_phi_sel	1	In	Block select from the PCU. When *pcu_phi_sel* is high both *pcu_adr* and *pcu_dataout* are valid.
pcu_rwn	1	In	Common read/not-write signal from the PCU.
pcu_adr[7:2]	6	In	PCU address bus. Only 6 bits are required to decode the address space for this block.
pcu_dataout[31:0]	32	In	Shared write data bus from the PCU.
phi_pcu_rdy	1	Out	Ready signal to the PCU. When *phi_pcu_rdy* is high it indicates the last cycle of the access. For a write cycle this means *pcu_dataout* has been registered by the block and for a read cycle this means the data on *phi_pcu_data* is valid.
phi_pcu_data[31:0]	32	Out	Read data bus to the PCU.

32.9.2 PHI sub-block partition
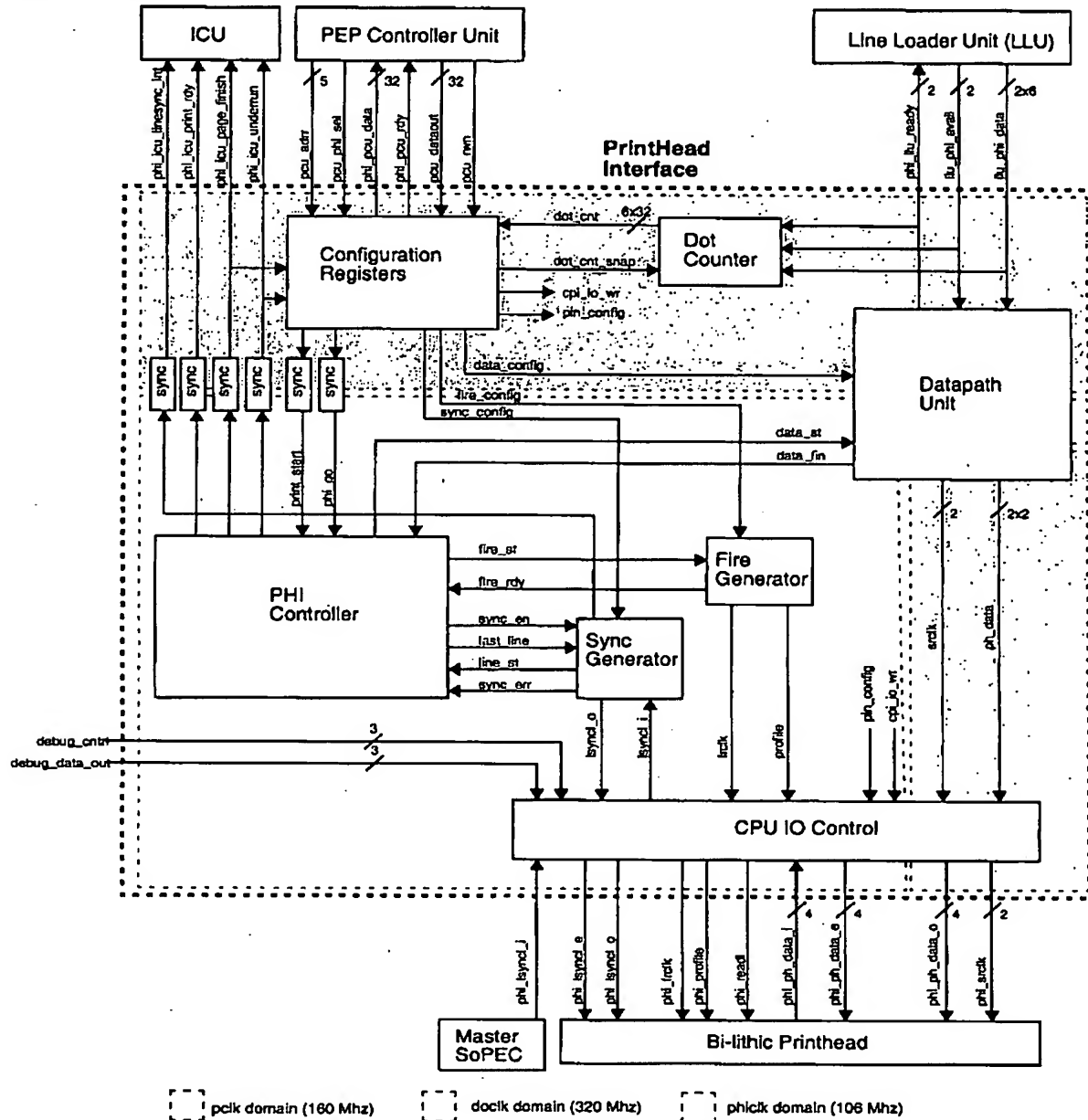
Figure 243. PHI block partition

32.9.3 Configuration registers

The configuration registers in the PHI are programmed via the PCU interface. Refer to section 21.8.2 on page 257 for a description of the protocol and timing diagrams for reading and writing registers in the PHI. Note that since addresses in SoPEC are byte aligned and the PCU only supports 32-bit register reads and writes, the lower 2 bits of the PCU address bus are not required to decode the address space for the PHI. When reading a register that is less than 32 bits wide zeros should be returned on the upper unused bit(s) of *phi_pcu_data*. Table 164 lists the configuration registers in the PHI

Table 164. PHI registers description

Address PHI_base+	Register	#bits	Reset	Description
Control Registers				
0x00	Reset	1	0x1	Active low synchronous reset, self de-activating. A write to this register will cause a PHI block reset.
0x04	Go	1	0x0	Active high bit indicating the PHI is programmed and ready to use. A low to high transition will cause PHI block internal state to reset. Will be automatically reset if a page finish or a buffer underrun is detected.
General Control				
0x08	PageLenLine	32	0x0000_0000	Specifies the number of dot lines in a page.
0x0c	PrintStart	1	0x0	A low to high transition triggers printing to start. Only active in Master Mode
0x10-0x14	DotMargin	2x16	0x0000	Specifies for each printhead IC, the width of the margin in dots divided by 2. 0 - Printhead IC Channel A 1 - Printhead IC Channel B
0x18-0x2C	DotCount[5:0]	6x32	0x0000_0000	Indicates the number of Dots used for a particular color, where N specifies a color from 0 to 5. Value valid after a write access to *DotCountSnap*
0x30	DotCountSnap	1	0x0	Write access causes the *AccumDotCount* values to be transferred to the *DotCount* registers. The *AccumDotCount* are reset afterwards.
0x34	PhiHeadSwap	1	0x0	Controls which signals are connected to printhead channels A and B 0 - Normal, specifies bit 0 is channel A, bit 1 is channel B 1 - Swapped, specifies bit 0 is channel B, bit 1 is channel A.
0x38	PhiMode	1	0x0	Indicates whether the PHI is operating in master or slave mode 0 - Slave Mode 1 - Master Mode
0x3C-0x40	PhiSerialOrder	2x1	0x0	Specifies the serialization order of dots before transfer to the printhead. Bus 0 - Printhead Channel A Bus 1 - Printhead Channel B A 0 indicates order ABC, while 1 indicates CBA

Table 164. PHI registers description

Address PHI base	Register	bits	Reset	Description
0x44-0x48	PrintHeadSize	2x16	0x0000	Specifies the number of non-margin dots in the printhead ICs. If margining is to be used then the configured *PrintHeadSize* should be adjusted by the dot margin value i.e. *PrintHeadSize* = (*Physical-PrintHeadSize* - (*DotMargin* * 2)). Bus 0 - Specifies printhead on Channel A Bus 1 - Specifies printhead on Channel B
CPU Direct PHI Control (See Table 161.)				
0x4C	PrintHeadCpuIn	5	0x00	PHI interface pins input status. Only active in direct CPU mode
0x50	PrintHeadCpuDir	5	0x00	PHI interface pins direction control. Only active in direct CPU mode
0x54	PrintHeadCpuOut	10	0x000	PHI interface pins output control. Only active in direct CPU mode
0x58	PrintHeadCpuCtrl	1	0x0	Control direct access CPU access to the PHI pins 0 - Normal Mode 1 - Direct CPU Control mode
Line Sync Control				
0x5C	LsyncLow	16	0x0000	Number of *phiclk* cycles *phi_lsync* should remain low.
0x60	LsyncHigh	16	0x0000	Number of *phiclk* cycles *phi_lsync* should remain high.
0x64	LsyncPre	16	0x0000	Number of *phiclk* cycles between *PrintStart* rising transition and the generated *phi_lsync* falling edge
0x68	LsyncMinPeriod	24	0x00_0 000	Minimum number of *phiclk* cycles between *Lsync* pulses. *Lsync* pulses of a shorter period will be rejected. Only used in slave mode.
0x6C	LsyncDeglitchCnt	4	0x3	Number of *phiclk* cycles to filter the incoming *Lsync* pulse from the master. Only used in slave mode.
0x70	LineSyncInterrupt	16	0x0000	Number of line syncs to occur before generating an interrupt. When set to zero interrupt is disabled.
Shift Register Control				
0x74	SrclkPre	14	0x0000	Number of *phiclk* cycles between *phi_lsync* falling edge and *phi_srclk* pulse generation, or printhead data transfer
0x78	SrclkPost	14	0x0000	Number of *phiclk* cycles allowed margin from last *srclk* pulse in a line to before next line sync
0x7C-0x80	PrintHeadRate[1:0]	2x16	0xFFFF	Specifies the active to inactive ratio of *phi_srclk* for the printhead ICs. A 1 indicates Active. Bus 0 - Printhead IC channel A Bus 1 - Printhead IC channel B
0x84	DotOrderMode	1	0x0	Specifies the dot transmit order to the printhead Channel A. Printhead Channel B is always the opposing order. 0 - Even before Odd dots 1 - Odd before Even dots
Fire Control				
0x88	ProfilePre	14	0x0000	Number of *phiclk* cycles *phi_lsync* falling edge and *phi_profile* pulse generation

Table 164. PHI registers description

Address PHI base+	Register	bits	Reset	Description
0x8C	ProfileLow	14	0x0000	Number of *phiclk* cycles *phi_profile* should remain low.
0x90	ProfileHigh	14	0x0000	Number of *phiclk* cycles *phi_profile* should remain high.
0x94	ProfileNum	16	0x0000	Number of profile pulses per line time.
0x98	FrclkPre	14	0x0000	Number of *phiclk* cycles *phi_lsync* falling edge and *phi_frclk* pulse generation
0x9C	FrclkLow	14	0x0000	Number of *phiclk* cycles *phi_frclk* should remain low.
0xA0	FrclkHigh	14	0x0000	Number of *phiclk* cycles *phi_frclk* should remain high.
0xA4	FrclkNum	16	0x0000	Number of *phi_frclk* pulses per line time.
Working Registers				
0xA8-0xAC	LineDotCnt	2x16	0x0000	Indicates the number of dot processed in the current line Bus 0 - Printhead Channel A Bus 1 - Printhead Channel B (Read Only Registers)
0xB0	LineCnt	32	0x0000 _0000	Indicates the number of lines processed in this page (Read Only Register)

The configuration registers in the PHI block are clocked at *pclk* rates but several blocks in the PHI are clocked by different and asynchronous clocks. Configuration values are not re-synchronized, it is therefore important that the *Go* register be set to zero while updating configuration values. This prevents logic from entering unknown states due to metastable clock domain transfers.

Some registers can be written to at any time such as the direct CPU control registers (*PrintHeadCpuIn*, *PrintHeadCpuDir*, *PrintHeadCpuOut* and *PrintHeadCpuCtrl*), the *Go* register and the *PrintStart* register. All registers can be read from at any time.

When one of the direct CPU control registers are written to the configuration registers block generates a 2 cycle pulse (*cpu_io_wr*) which is used to transfer the pin control signals from the *pclk* domain to the *phiclk* domain. The *cpu_io_wr* signal is a delayed version of the write enable from the CPU.

32.9.4   Dot counter

The dot counter keeps a running count of the number of dots fired for each color plane. The counters are 32 bits wide and will saturate. When the CPU wants to read the dot count for a particular color plane it must write to the *DotCountSnap* register. This causes all 6 running counter values to be transferred to the *DotCount* registers in the configuration registers block. The running counter values are reset.

```
// reset if being snapped
if (dot_cnt_snap == 1) then{
    dot_count[5:0]        = accum_dot_count[5:0]
    accum_dot_count[5:0]  = 0
    }
// update the counts
for (color=0;color < 6;color++) {
    if (accum_dot_count[color] != 0xffff_ffff) {
        // data valid, first dot stream
        data_valid = ((phi_llu_ready[0] == 1) AND (llu_phi_avail[0] == 1))
        if ((data_valid == 1) AND (llu_phi_data[0][color] == 1)) then
            accum_dot_count[color] ++
```

```
// data valid, second dot stream
data_valid = ((phi_llu_ready[1] == 1) AND (llu_phi_avail[1] == 1))
if ((data_valid == 1) AND (llu_phi_data[1][color] == 1)) then
    accum_dot_count[color] ++
}
}
```

32.9.5 Sync generator

The sync generator logic has two modes of operation, master and slave mode. In master mode (configured by the *PhiMode* register) it generates the *lsyncl_o* output based on configured values and control triggers from the PHI controller. In slave mode it de-glitches the incoming *lsyncl_i* signal, and filters the *lsyncl* signal with the minimum configured period.
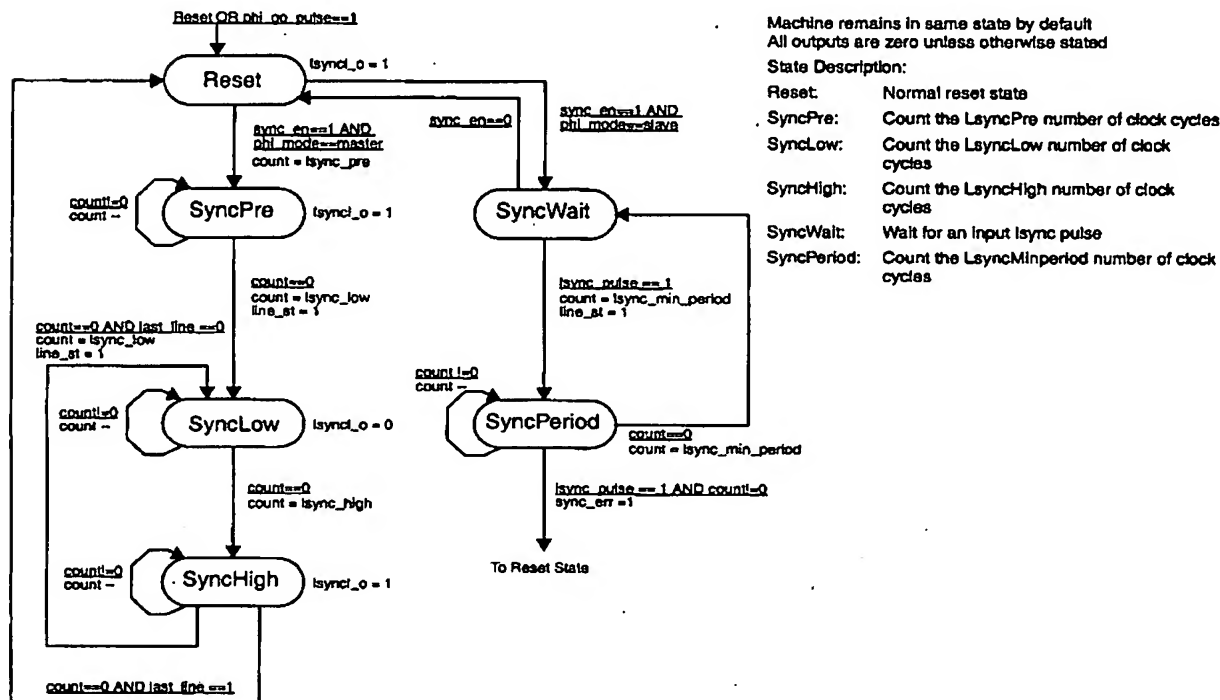
Figure 244. Sync generator state diagram

After reset or a pulse on *phi_go_pulse* the machine returns to the *Reset* state, regardless of what state it's currently in.

The state machine waits until it's enabled (*sync_en*==1) by the PHI controller state machine. When enabled it can proceed to the *SyncPre* or *SyncWait* depending on whether the state machine is configured in master or slave mode. In master mode it generates the *lsyncl* pulses, in slave mode it receives and filters the *lsyncl* pulses from the master sync generator.

On transition to the *SyncPre* state a counter is loaded with the *LsyncPre* value, and while in the *SyncPre* the counter is decremented. When the count is zero the machine proceeds to the *SyncLow* state pulsing the *line_st* signal on transition and loading the counter with *LsyncLow* value. This indicates to the PHI controller the line start aligned to the *lsyncl* negative edge.

The machine waits in the *SyncLow* state until the counter has decremented to zero. It proceeds to the *SyncHigh* state and counts *LsyncHigh* number of cycles. While in *LsyncLow* state the *lsyncl_o* output is set to 0 and in *SyncHigh* the *lsyncl_o* output is set to 1.

When the count is zero and the current line is not the last (*last_line* == 0), the machine returns to the *SyncLow* state to begin generating a new line sync pulse. The transition pulses the *line_st* signal to the PHI controller.

The loop is repeated until the current line is the last (*last_line* ==1), and the machine returns to the *Reset* state to wait for the next page start.

In slave mode the state machine proceeds to the *SyncWait* state when enabled. It waits in this state until a *lsync_pulse* is received from the input de-glitch circuit. When a pulse is detected the machine jumps to the *SyncPeriod* state and begins counting down the *LsyncMinPeriod* number of clock cycles before returning to the *SyncWait* state. On transition from the *SyncWait* to the *SyncPeriod* state the *line_st* signal to the PHI controller is pulsed to indicate the line start. While in the *SyncPeriod* state if a *lsync_pulse* is detected the state machine will signal a sync error (via *sync_err*) to the PHI controller and cause a buffer underrun interrupt.

32.9.5.1 Lsyncl input de-glitch

The *lsync_i* input is considered an asynchronous input to the PHI, and is passed through a synchronizer to reduce the possibility of metastable states occurring before being passed to the de-glitch logic.

The input de-glitch logic rejects input states of duration less than the configured number of clock cycles (*lsync_deglitch_cnt*), input states of greater duration are reflected on the output, and are negative edge detected to produce the *lsync_pulse* signal to the main generator state machine. The counter logic is given by

```
if ( lsync_i != lsync_i_delay) then
   cnt        = lsync_deglitch_cnt
   output_en = 0
elsif (cnt == 0 ) then
   cnt        = cnt
   output_en = 1
else
   cnt --
   output_en = 0
```
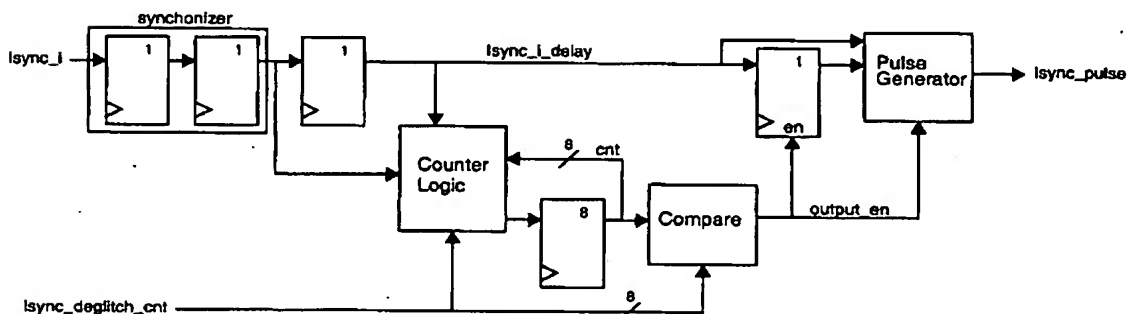

Figure 245. Line sync de-glitch RTL diagram

### 32.9.5.2 Line Sync Interrupt logic

The line sync interrupt logic counts the number of line syncs that occur (either internally or externally generated line syncs) and determines whether to generate an interrupt or not. The number of line syncs it

counts before an interrupt is generated is configured by the *LineSyncInterrupt* register. The interrupt is disabled if *LineSyncInterrupt* is set to zero.

```
// implement the interrupt counter
if (phi_go_pulse ==1) then
   line_count = 0
elsif (line_st == 1) AND (line_count == 0)) then
      line_count = linecount_int
elsif ((line_st == 1) AND (line_count != 0)) then
      line_count --
// determine when to pulse the interrupt
if (linesync_int == 0 ) then // interrupt disabled
   phi_icu_linesync_int = 0;
elsif ((line_st == 1) AND (line_count == 1)) then
   phi_icu_linesync_int = 1
```

32.9.6   Fire generator

The fire generator block creates the signal profile for the *phi_frclk* and *phi_profile* signals to the printhead. The profile is based on configured values and is timed in relation to the *fire_sync* pulse from the PHI controller block.
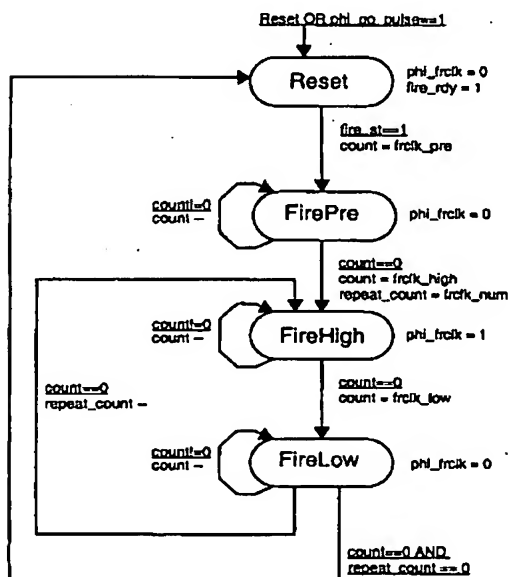
Figure 246. Fire generator state diagram

The fire generator consists of 2 identical state machines for creating the *phi_frclk* and *phi_profile* signals respectively.

The machine is reset to the *Reset* state when *phi_go_pulse* ==1 or the reset is active, regardless of the current state.

The machine waits in the reset state until it receives a *fire_st* pulse from the PHI controller. The controller will generate a *fire_st* pulse at the beginning of each dot line. On the state transition the cycle counter is loaded with the *FrclkPre* value and the repeat counter is loaded with the *FrclkNum* value.

The state machine waits in the *FirePre* state until the cycle counter is zero, after which it jumps to the *Fire-High* state and loads the cycle counter with *FrclkHigh* value. Again the state machine waits until the count is zero and then proceeds to the *FireLow* state. On transition the cycle counter is loaded with the *FireLow* value. The state machine waits in the *FireLow* state while the cycle counter is decremented.

When the cycle counter reaches zero and the *repeat_count* is non-zero, the *repeat_count* is decremented, the cycle counter is loaded with the *FrclkHigh* value and the state machine jumps to the *FireHigh* state to repeat the *phi_frclk* generation cycle. The loop is repeated until the *repeat_count* is zero. In such cases the state machine goes to the reset state and waits for the next *fire_st* pulse.

When in the *Reset* state the *fire_rdy* signal is active to indicate to the controller that the fire generator is ready.

32.9.7 PHI controller

The PHI controller is responsible for controlling all functions of the PHI block on a line by line basis. It controls and synchronizes the sync generator, the fire generator, and datapath unit, as well as signalling

back to the CPU the PHI status. It also contains a line counter to determine when a full page has completed printing.
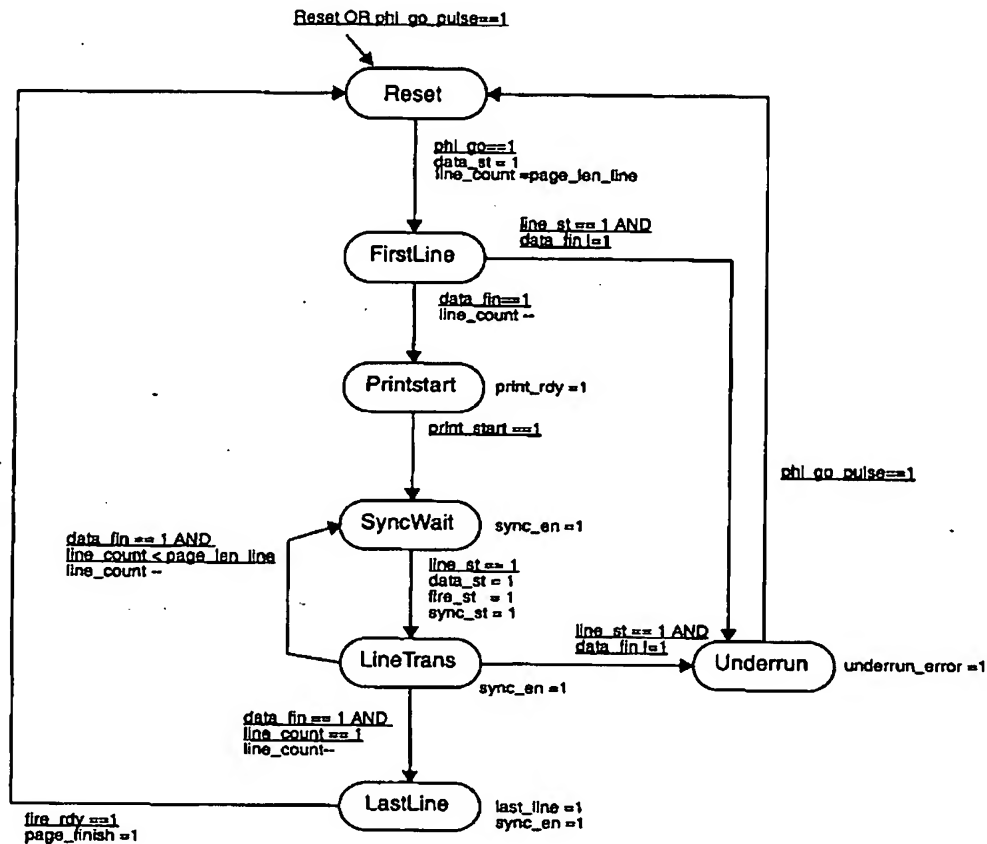

Figure 247. PHI controller state machine

The PHI controller state machine is reset to *Reset* state by a reset or *phi_go_pulse* $= 1$.

It will remain in reset until the block is enabled by *phi_go* $= 1$. Once enabled the state machine will jump to the *FirstLine* state, trigger the transfer of one line of data to the printhead (*data_st* $= 1$) and the line counter will be initialized to the page length (*PageLenLine*). Once the line is transferred (*data_fin* from the datapath unit) the machine will go to *Printstart* state and signal the CPU using an interrupt that the PHI is ready to begin printing (*phi_icu_print_rdy*). The line counter will also be decremented. It will then wait in the *Printstart* state until the CPU acknowledges the print ready signal and enables printing by writing to the *PrintStart* register.

The state machine proceeds to the *SyncWait* state and waits for a line start condition (*line_st* $=1$). The line start condition is different depending on whether the PHI is configured as being in a master or slave SoPEC (the *PhiMode* register). In either case the sync generator determines the correct line start source and signals the PHI controller via the *line_st* signal. Once received the machine proceeds to the *LineTrans* state, with the transition triggering the fire generator to start (*fire_st*), the datapath unit to start (*data_st*) and the sync generator to start (*sync_st*).

While in the *LineTrans* state the fire, sync and datapath unit will be producing line data. When finished processing a line the datapath unit will assert the line finished (*line_fin*) signal. If the line counter is not equal to 1 (i.e. not the last line) the state machine will jump back to the *SyncWait* state and wait for the start condition for the next line. The line counter will be decremented. If the line counter is one then the machine will proceed to the *LastLine* state.

The *LastLine* state generates one more line of fire pulses to print the last line held in the shift registers of the printhead. Once complete (*fire_fin* ==1) the state machine returns to the reset state and waits for the next page of data. On page completion the state machine generates a *phi_icu_page_finish* interrupt to signal to the CPU that the page has completed, the *phi_icu_page_finish* will also cause the *Go* register to reset automatically.

While the state machine is in the *LineTrans* state (or in *FirstLine* state and the PHI is in slave mode) and waiting for the datapath unit to complete line processing, it is possible (e.g. an excessive PEP stall) that a new line start condition occurs but the datapath unit is not ready. In this case an underrun error is generated. The state machine goes to the *Underrun* state and generates a *phi_icu_underrun* interrupt to the CPU. The PHI cannot recover from a buffer underrun error, the CPU must reset the PEP blocks and restart printing. The *phi_icu_underrun* will also cause the *Go* register to reset automatically.

32.9.8 CPU IO control

The CPU IO control block is responsible for accepting CPU direct IO control signals from the configuration registers (at *pclk* frequency) and transferring them to *phiclk* frequency. It also accepts the input signals from the printhead and re-synchronizes them to the *pclk* domain, and debug signals from the RDU and muxes them to output pins.

Table 161 contains the direct mapping of configuration registers to printhead IO pins. Direct CPU control is enabled only when *PrintHeadCpuCtrl* is set to one. In normal operation (i.e. *PrintHeadCpuCtrl* == 0) the printhead data pins are always in output mode (*phi_ph_data_e* == 1), the *phi_lsyncl* will be in output if the SoPEC is the master, i.e. *phi_lsyncl_e* = *phi_mode*, and *readl* will be set high.

The pseudocode for the CPU IO control is:

```
if (printhead_cpu_ctrl == 1) then  // CPU access enabled
    // outputs
    phi_ph_data_o[0][1:0]  = printhead_cpu_out[1:0]
    phi_ph_data_o[1][1:0]  = printhead_cpu_out[3:2]
    phi_lsyncl_o           = printhead_cpu_out[4]
    phi_readl              = printhead_cpu_out[5]
    phi_srclk[1:0]         = printhead_cpu_out[7:6]
    phi_frclk              = printhead_cpu_out[8]
    phi_profile            = printhead_cpu_out[9]
    // direction control
    phi_ph_data_e[0][1:0]  = printhead_cpu_dir[1:0]
    phi_ph_data_e[1][1:0]  = printhead_cpu_dir[3:2]
    phi_lsyncl_e           = printhead_cpu_dir[4]
    // input assignments
    printhead_cpu_in[1:0]  = synchronize(phi_ph_data_i[0][1:0])
    printhead_cpu_in[3:2]  = synchronize(phi_ph_data_i[1][1:0])
    printhead_cpu_in[5]    = synchronize(phi_lsyncl_i[0][1:0])
else  // normal connections
    // outputs
    phi_ph_data_o[0][1:0]  = ph_data[0][1:0]
    phi_ph_data_o[1][1:0]  = ph_data[1][1:0]
    phi_lsyncl_o           = lsync_o
    phi_readl              = 1
    phi_srclk[1:0]         = srclk[1:0]
    phi_frclk              = frclk
    phi_profile            = profile
    // direction control
```

```
phi_ph_data_e[0][1:0] = 0x3
phi_ph_data_e[1][1:0] = 0x3
phi_lsyncl_e             = phi_mode     // depends on Master or Slave mode
// inputs
lsyncl_i                 = phi_lsync_i // connected regardless
// debug overrides any other connections
if (debug_cntrl[0] == 1) then
   phi_frclk             = debug_data_out[0]
   phi_readl             = pclk
if (debug_cntrl[1] == 1) then
   phi_profile           = debug_data_out[1]
if (debug_cntrl[2] == 1) then
   phi_lsyncl_o          = debug_data_out[2]
   phi_lsyncl_e          = 1
```

The debug signalling is controlled by the RDU block (see Section 11.8 Realtime Debug Unit (RDU)), the IO control in the PHI muxes debug data onto the PHI pins based on the control signals from the RDU.
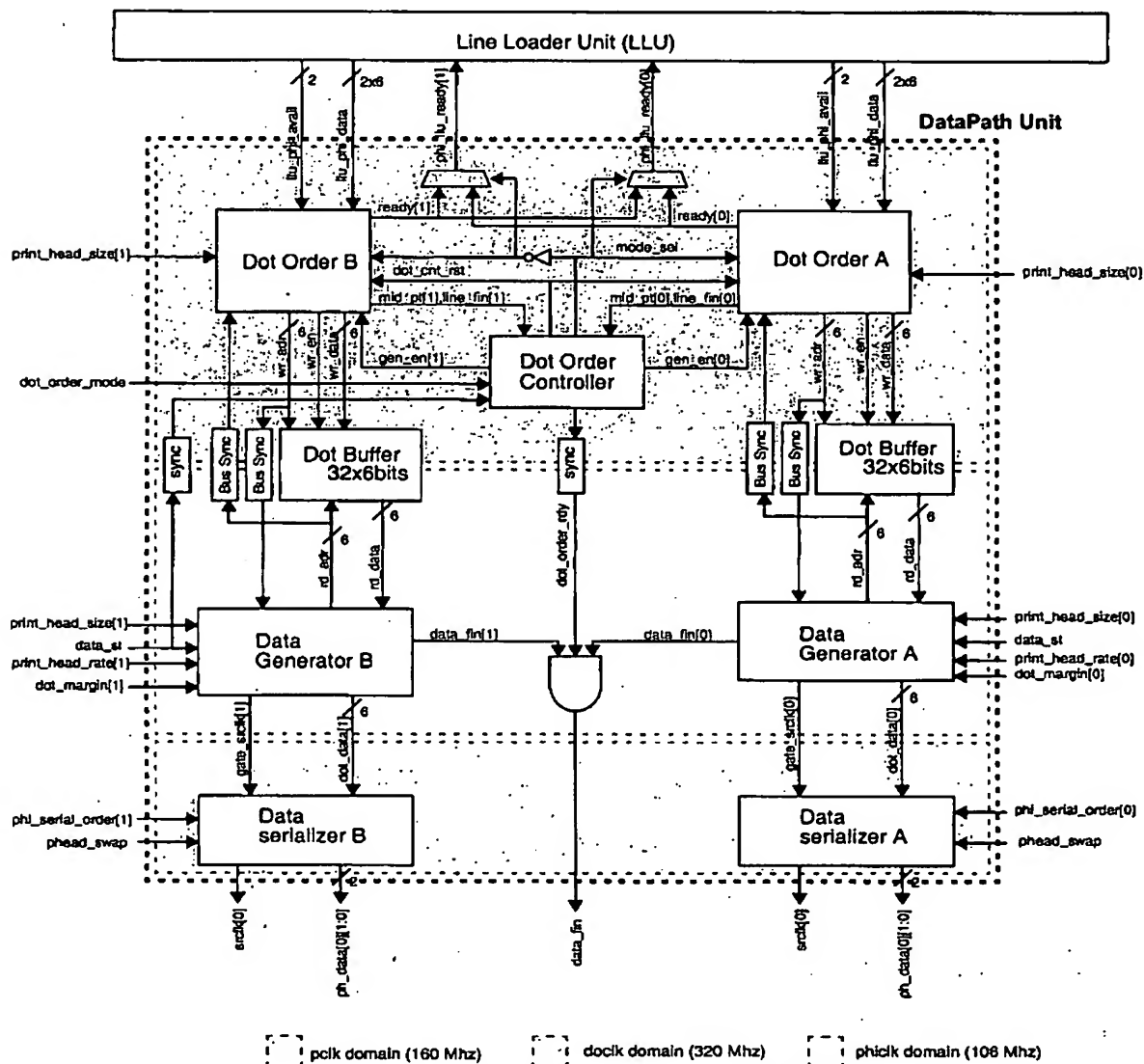
32.9.9   Datapath Unit

Figure 248. Datapath Unit partition
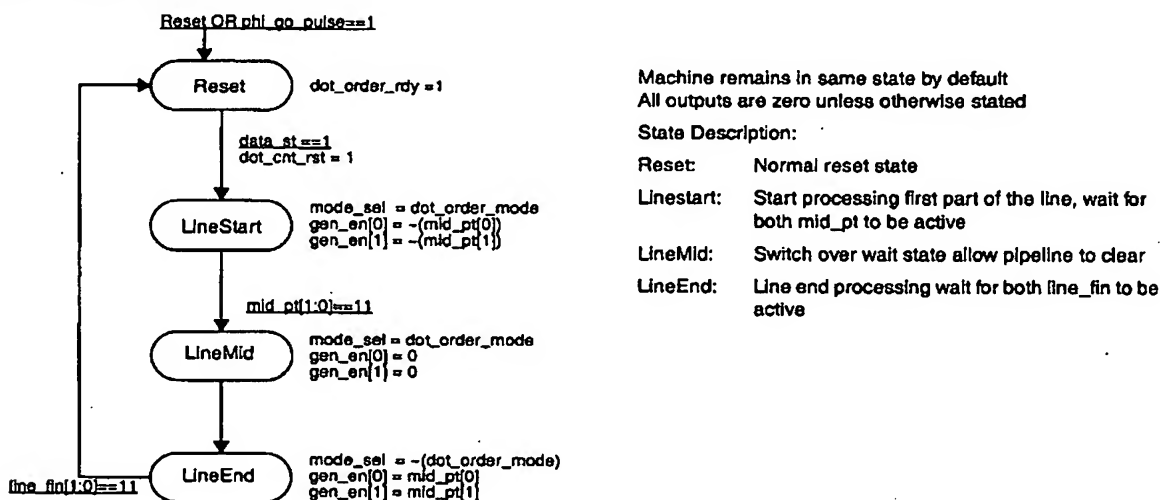
32.9.10 Dot order controller

Figure 249. Dot Order controller state diagram

The dot order controller is responsible for controlling the dot order blocks. It monitors the status of each block and determines the switch over point, at which the connections from odd and even dot streams to printhead channels are swapped.

The machine is reset to the *Reset* state when *phi_go_pulse* == 1 or the reset is active. The machine will wait until it receives a *data_st* pulse from the PHI controller before proceeding to the *LineStart* state. On the transition to the *LineStart* state it will reset the dot counter in each dot order block via the *dot_cnt_rst* signal.

While in the *LineStart* state both dot order blocks are enabled (*gen_en*==1). The dot order blocks process data until each of them reach their mid point. The mid point of a line is defined by the configured printhead size (i.e. *print_head_size*). When a dot order block reaches the mid point it immediately stops processing and waits for the remaining dot order block. When both dot order blocks are at the mid point (*mid_pt* == 11) the controller clocks through the *LineMid* state to allow the pipeline to empty and immediately goes to *LineEnd* state.

In the *LineEnd* state the *mode_sel* is switched and the dot order blocks re-enabled, in this state the dot order blocks are reading data from the opposite LLU dot data stream as in *LineStart* state. The controller remains in the *LineEnd* state until both dot order blocks have processed a line i.e. *line_fin* == 11.

On completion of both blocks the controller returns to the *Reset* state and again awaits the next *data_st* pulse from the PHI controller. When in *Reset* state the machine signals the PHI controller that it's ready to begin processing dot data via the *dot_order_rdy* signal.

The dot order controller selects which dot streams should feed which printhead channels. The order can be changed by configuring the *DotOrderMode* register. In all cases Channel A and Channel B must be in opposing dot order modes. Table 158 shows the possible modes of operation.

Table 165. Mode selection in Dot order controller.

Channel	Mode sel	DotOrderMode	Dot transmit order
A	0	0	Even before Odd (EBO mode), even dot stream feeds Channel A printhead, first half line.
	0	1	Odd before Even (OBE mode), odd dot stream feeds Channel A printhead, first half line.
	1	0	Even before Odd (EBO mode), even dot stream feeds Channel A printhead, second half line.
	1	1	Odd before Even (OBE mode), odd dot stream feeds Channel A printhead, second half line.
B	0	0	Odd before Even (OBE mode), odd dot stream feeds Channel B printhead, second half line
	0	1	Even before Odd (EBO mode), even dot stream feeds Channel B printhead, second half line.
	1	0	Odd before Even (OBE mode), odd dot stream feeds Channel B printhead, first half line.
	1	1	Even before Odd (EBO mode), even dot stream feeds Channel B printhead, first half line.

32.9.10.1 Dot order unit

The dot order control accepts dot data from either dot stream from the LLU and writes the dot data into the dot buffer. It has two modes of operation, odd before even (OBE) and even before odd (EBO). In the OBE mode data from the odd stream dot data is accepted first then even, in EBO mode it's vice versa. The mode is configurable by the *DotOrderMode* register.

The dot order unit maintains a dot count that is decremented each time a new dot is received from the LLU. The dot order controller resets the dot counter to the *print_head_size[15:0]* at the start of a new line via the *dot_cnt_rst* signal. The dot count is compared with the printhead size (*print_head_size[15:0]* divided by 2) to determine the mid point (*mid_pt*) and the line finish point (*line_fin*) when the dot counter is zero.

The mid point is defined as the half the number of dots in a particular printhead, and is given by the *print_head_size* bus.

```
// define the mid point
if (dot_cnt[15:0] == print_head_size[15:1] )then
    mid_pt = 1
else
    mid_pt = 0
```

The dot order unit logic maintains the dot data write pointer. Each time a new dot is written to the dot buffer the write pointer is incremented. The fill level of the dot buffer is determined by comparing the read and write pointers. The fill level is used to determine when to backpressure the LLU (*ready* signal) due to the dot buffer filling. A suitable threshold value is determined to allow for the full LLU pipeline to empty into the dot buffer.

The dot order stalling control is given by:

```
// determine the ready/avail signal to use, based on mode select
if (mode_sel == 1) then
    dot_active = llu_phi_avail[0] AND ready
    wr_data    = llu_phi_data[0]
```

```
else
   dot_active = llu_phi_avail[1] AND ready
   wr_data    = llu_phi_data[1]
// update the counters
if (dot_active == 1) then (
   wr_en = 1
   wr_adr ++
   if (dot_cnt == 0) then
      dot_cnt = print_head_size
   else
      dot_cnt--
   )
```

The dot writer needs to determine when to stall the LLU dot data stream. A number of factors could stall the dot stream in the LLU such as buffer filling, waiting for the mid point, waiting for the line finish or the dot order controller is waiting for the line start condition from the PHI controller.

The stall logic is given by:
```
// determine when to stall the LLU generator
fill_level = wr_adr - rd_adr
if (fill_level > (32 - THRESHOLD ))then   // THRESHOLD is open value TBD
   ready = 0                              // buffer is close to full
elsif ( gen_en == 0) then
   ready = 0                              // stalled by the datapath controller
else
   ready = 1                              // everything good no stall
```

32.9.10.2 Data generator
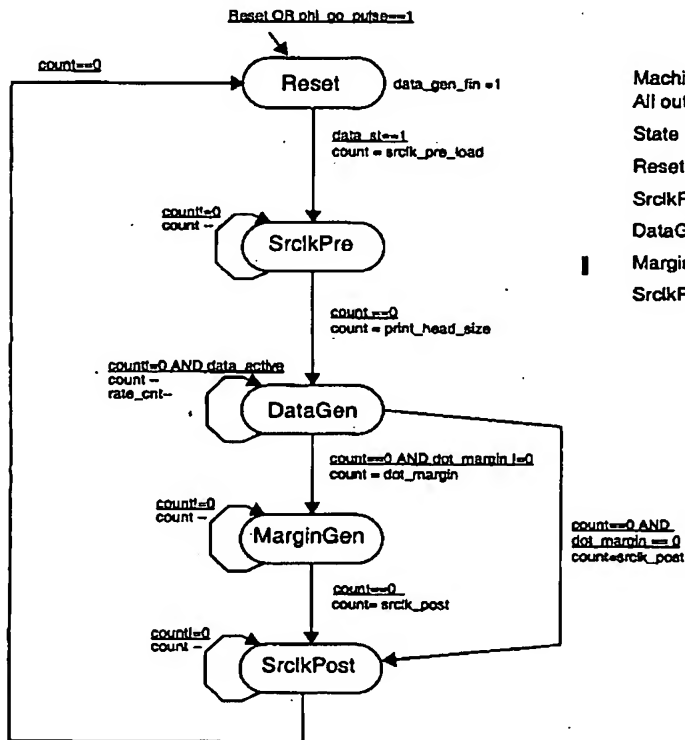
Figure 250. Data generator state diagram

The data generator block reads data from the dot buffer and feeds dot data to the printhead at a configured rate (set by the *PrintheadRate*). It also generates the margin zero data and aligns the dot data generation to the synchronization pulse from the PHI controller.

The data generator controller waits in *Reset* state until it receives a line start pulse from the PHI controller (*data_st* signal). Once a start pulse is received it proceeds to the *SrclkPre* state loading a counter with the *SrclkPre* value. While in this state it decrements the counter. No data is read or output at this stage. When the count is zero the machine proceeds to the *DataGen* state.

On transition it loads the counter with the printhead size (*print_head_size*). If margining is to be used then the configured *print_head_size* should be adjusted by the dot margin value i.e. *print_head_size = (physical_print_head_size - (dot_margin * 2))*.

While in *DataGen* state data is read from the dot buffer and output to the printhead. The counter will decrement for every dot data word transferred. The exact rate is dictated by the dot buffer fill levels and the configured printhead rate (*PrintheadRate*).

The generator determines the rate by incrementing a rate counter (*rate_cnt*) while in the *DataGen* state. The rate counter is allowed to wrap normally. If the bit selected by the *rate_cnt* in the *print_head_rate* bus is one data is transferred, otherwise the cycle is skipped. If the *PrintHeadRate* is set to all zeros then no data will ever get transferred. The pseudo-code for the *DataGen* state is given by:

```
// increment the rate count
rate_cnt ++
// determine if data should be read
```

```
// first determine if data is available in buffer
if (rd_adr != wr_adr ) .then
   if (print_head_rate[rate_cnt] == 1 ) then
      dot_active = 1
      gate_srclk = 1
      rd_adr ++
      dot_data   = rd_data
      count --
   else
      dot_active = 0
      gate_srclk = 0
else
   dot_active = 0
   gate_srclk = 0
```

When the counter reaches zero the state machine will jump to the *MarginGen* state if the configured margin value is non-zero, otherwise it will jump directly to the *SrclkPost* state. On transition to *MarginGen* state it loads the cycle counter with the *dot_margin* value, and begins to count down. While in the *MarginGen* state the data generator logic block writes dot data to the printhead but does not read from the dot buffers. It creates zero dot data words for the margin duration.

When the counter reaches zero the machine jumps to the *SrclkPost* state, loads the clock counter with the *SrclkPost* value and decrements. When the count is finished the state machine returns to the *Reset* and awaits the next start pulse. Should a line sync arrive before the data generators have completed (*data_fin* signal) the PHI controller will detect a print error and stall the PHI interface.

32.9.10.3 Data serializer

The data serializer block converts 6-bit dot data at *phiclk* rates (nominally 106 MHz) to 2-bit data at *doclk* rates (nominally 320 MHz).
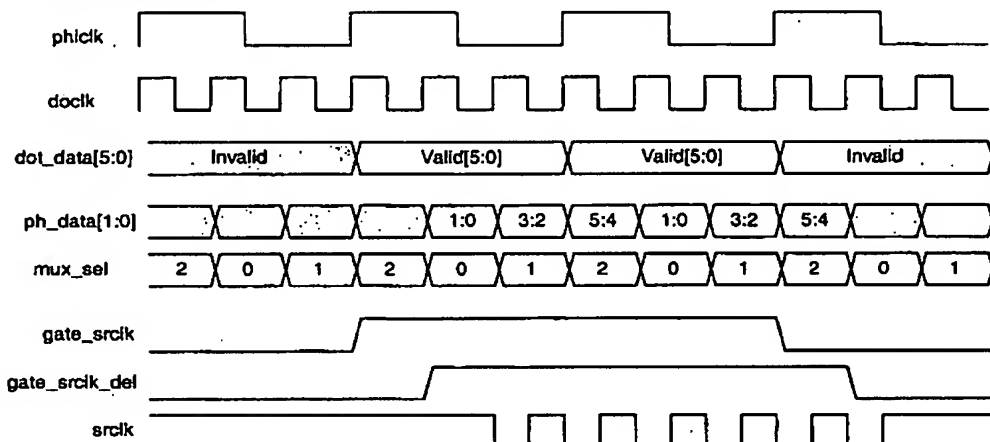

Figure 251. Data serializer timing

The *srclk* is only active when data is available for transfer to the printhead, as enabled by the *gate_srclk* signal. The data rate mechanism in the data generator block will mean that data is not transferred to the printhead on every *phiclk* cycle. Both the *dot_data* and *gate_srclk* signals are clocked out by the *phiclk* and can only change on the rising of *phiclk*.

The data serializer block allows easy separation of clock gating and clock to logic structures from the rest of the PHI interface. All registers in the block are clocked at *doclk* rates.
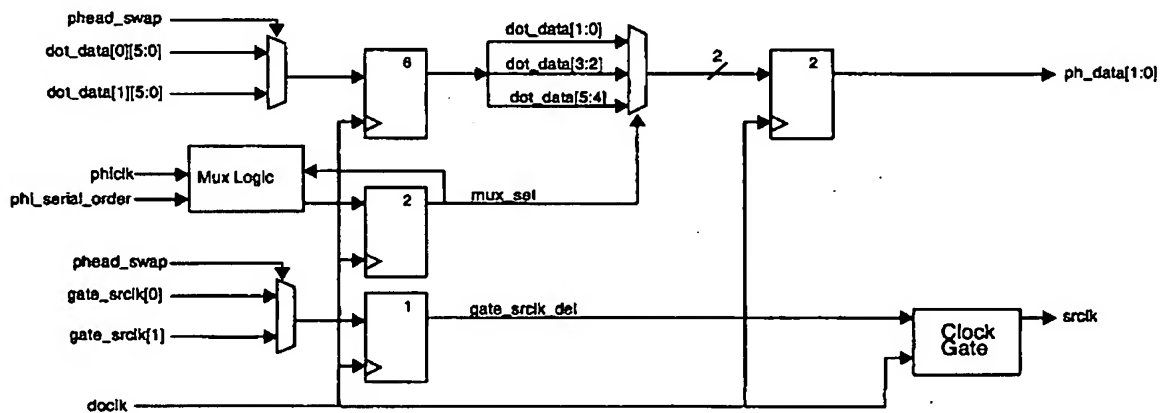

Figure 252. Data serializer RTL Diagram

The mux logic determines which data bits from the *dot_data* bus should be selected for output on the *ph_data* to the printhead. The selection is dependent on the *phiclk* edge.

```
if (phiclk == 1) then
    mux_sel = 1
elsif ( mux_sel == 2 ) then
    mux_sel = 0
else
    mux_sel++
```

The dot data serialization order can be configured by *PhiSerialOrder* register. If the *PhiSerialOrder* is zero the order is *dot[1:0]*, then *dot[3:2]* then *dot[5:4]*. If the register is one then the order is *dot[5:4]*, *dot[3:2]*, *dot[1:0]*.

PACKAGE AND TEST

33 Test Units

33.1 JTAG INTERFACE

A standard JTAG (Joint Test Action Group) Interface is included in SoPEC for Bonding and IO testing purposes. The JTAG port will provide access to all internal BIST (Built In Self Test) structures.

33.2 SCAN TEST I/O

The SoPEC device will require several test IO's for running scan tests. In general scan in and scan out pins will be multiplexed with functional pins.

33.3 ANALOG TEST UNITS

33.3.1 USB PHY Testing

The USB phy analog macro, will contain built-in in test structure, which can be access by either the CPU or through the JTAG port.

33.3.2 Embedded PLL Testing

The embedded clock generator PLL will require test access from JTAG port.

34 SoPEC Pinning and Package

34.1 OVERVIEW

It is intended that the SoPEC package be a 100 pin LQFP. Any spare pins in the package may be used by increasing the number of available GPIO pins or adding extra power and ground pin. The pin list shows the minimum pin requirement for the SoPEC device.

Table 166. SoPEC Pin List

Pin Name	#pins	Dir	Type	Volt	Internal Name	Description
Clocks and resets						
xtalin	1	I	TBD	N/A	xtalin	Crystal input pin
xtalout	1	O	TBD	N/A	xtalout	Crystal output pin
reset_n	1	I	LVTTL	2.5v	reset_n	Asynchronous active low reset
Printhead Interface						
ph_data[0][0]	2	O	LVDS	3.3v	phi_ph_data_o[0][0]	Dot data for colors 0-2 for Printhead 0. Using differential signalling
		I	LVTTL	3.3v	phi_ph_data_i[0]	Input mode bit used for nozzle test result printhead 0
ph_data[0][1]	2	O	LVDS	3.3v	phi_ph_data_o[0][1]	Dot data for colors 3-5 for Printhead 0. Using differential signalling
		I	LVTTL	3.3v	phi_ph_data_i[1]	Input mode bit used for temperature data printhead 0
ph_data[1][0]	2	O	LVDS	3.3v	phi_ph_data_o[1][0]	Dot data for colors 0-2 for Printhead 1. Using differential signalling
		I	LVTTL	3.3v	phi_ph_data_i[1]	Input mode bit used for nozzle test result printhead 1
ph_data[1][1]	2	O	LVDS	3.3v	phi_ph_data_o[1][1]	Dot data for colors 3-5 for Printhead 1. Using differential signalling
		I	LVTTL	3.3v	phi_ph_data_i[1]	Input mode bit used for temperature data printhead 1
srclk[0]	2	O	LVDS	3.3v	phi_srclk[0]	Differential dot data shift clock for print head 0
srclk[1]	2	O	LVDS	3.3v	phi_srclk[1]	Differential dot data shift clock for print head 1
readl	1	O	LVTTL	3.3v	phi_readl	Common Print head mode control
frclk	1	O	LVTTL	3.3v	phi_frclk	Common Fire pattern shift clock, needs to toggle once per fire cycle
profile	1	O	LVTTL	3.3v	phi_profile	Common Pulse profile for all colors
lsyncl	1	O	LVTTL	3.3v	phi_lsyncl_o	Line Sync output from Master to Slaves
		I	LVTTL	3.3v	phi_lsyncl_i	Line Sync input to Slaves from Master
USB Connections						
usbd	2	I/O	Differential	3.3v	Direct Phy Connection	USB differential data
JTAG						
tdo	1	O	CMOS	2.5v	tdo	JTAG Test data out port
tms	1	I	CMOS	2.5v	tms	JTAG Test mode select
tdi	1	I	CMOS	2.5v	tdi	JTAG Test data in port
tck	1	I	CMOS	2.5v	tck	JTAG Test access port clock
General Purpose IO						

Table 166. SoPEC Pin List

Pin Name	# pins	Dir	Type	Volt	Internal Name	Description
gpio[3:0]	4	O	CMOS	2.5v	gpio_o[3:0]	Motor control pins / general purpose Output
		I	CMOS	2.5v	gpio_i[3:0]	General purpose Input
gpio[7:4]	4	O	High Drive CMOS	2.5v	gpio_o[7:4]	LED driver pins / general purpose Output
		I	CMOS	2.5v	gpio_i[7:4]	General purpose Input
gpio[11:8]	4	O	Open collector	2.5v	gpio_o[11:8]	LSS interface pins / general purpose Output
		I	CMOS	2.5v	gpio_i[11:8]	LSS interface pins / general purpose Input
gpio[13:12]	2	O	CMOS	2.5v	gpio_o[13:12]	ISI interface pins / general purpose Output
		I	CMOS	2.5v	gpio_i[13:12]	ISI interface pins / general purpose Input
Test Pins						
test_enable	1	I	CMOS	2.5v	TBD	Test Enable
generic_test	5	I/O	CMOS	2.5v	TBD	Generic test pin, function undefined
Total Signal Pins	45					
Power Pins						
gnd	18	I	Power	N/A	gnd	gnd
vdd	10	I	Power	N/A	vdd	vdd 1.5v, core voltage
vdd250	3	I	Power	N/A	vdd250	vdd 2.5v, IO voltage
vdd330	5	I	Power	N/A	vdd330	vdd 3.3v, IO voltage
Total Pins	81					

MEMJET PRINTHEAD

35 Memjet Printhead

This section is quoted verbatim from SoPEC/MoPEC Bilithic Printhead Reference document [10].

35.1 BACKGROUND

Silverbrook's bilithic Memjet^{TM} printheads are the target printheads for printing systems which will be controlled by SoPEC and MoPEC devices.

This document presents the format and structure of these printheads, and describes the their possible arrangements in the target systems. It also defines a set of terms used to differentiate between the types of printheads and the systems which use them.

35.2 COMPANION DOCUMENTS

Currently, this document is only concerned with the structure of the printheads and their systems, with regard to the way in which dot data is loaded.

Refer to the Bilithic Printhead Specification [2] for the complete description of the functionality of these devices.

This document relies on certain definitions and details presented in Bilithic Printhead Specification [2].

35.3 DEFINITIONS

This document presents terminology and definitions used to describe the bilithic printhead systems. These terms and definitions are as follows:
- Printhead Type - There are 3 parameters which define the type of printhead used in a system:
  - Direction of the data flow through the printhead (clockwise or anti-clockwise, with the printhead shooting ink down onto the page).
  - Location of the left-most dot (upper row or lower row, with respect to V_+).
  - Printhead footprint (type A or type B, characterized by the data pin being on the left or the right of V_+ where V_+ is at the top of the printhead).
- Printhead Arrangement - Even though there are 8 printhead types, each arrangement has to use a specific pairing of printheads, as discussed in Section 35.4. This gives 4 pairs of printheads. However, because the paper can flow in either direction with respect to the printheads, there are a total of eight possible arrangements, e.g. Arrangement 1 has a Type 0 printhead on the left with respect to the paper flow, and a Type 1 printhead on the right. Arrangement 2 uses the same printhead pair as Arrangement 1, but the paper flows in the opposite direction.
- Color 0 is always the first color plane encountered by the paper.
- Dot 0 is defined as the nozzle which can print a dot in the left-most side of the page.
- The Even Plane of a color corresponds to the row of nozzles that prints dot 0.

Note that throughout this document, where the various printheads and systems are presented, the printheads always shoot ink down onto the page.

Figure 253 shows the 8 different possible printhead types. Type 0 is identical to the Right Printhead presented in Figure 3 in [2], and Type 1 is the same as the Left Printhead as defined in [2].

While the printheads shown in Figure 253 look to be of equal width (having the same number of nozzles) it is important to remember that in a typical system, a pair of unequal sized printheads may be used.
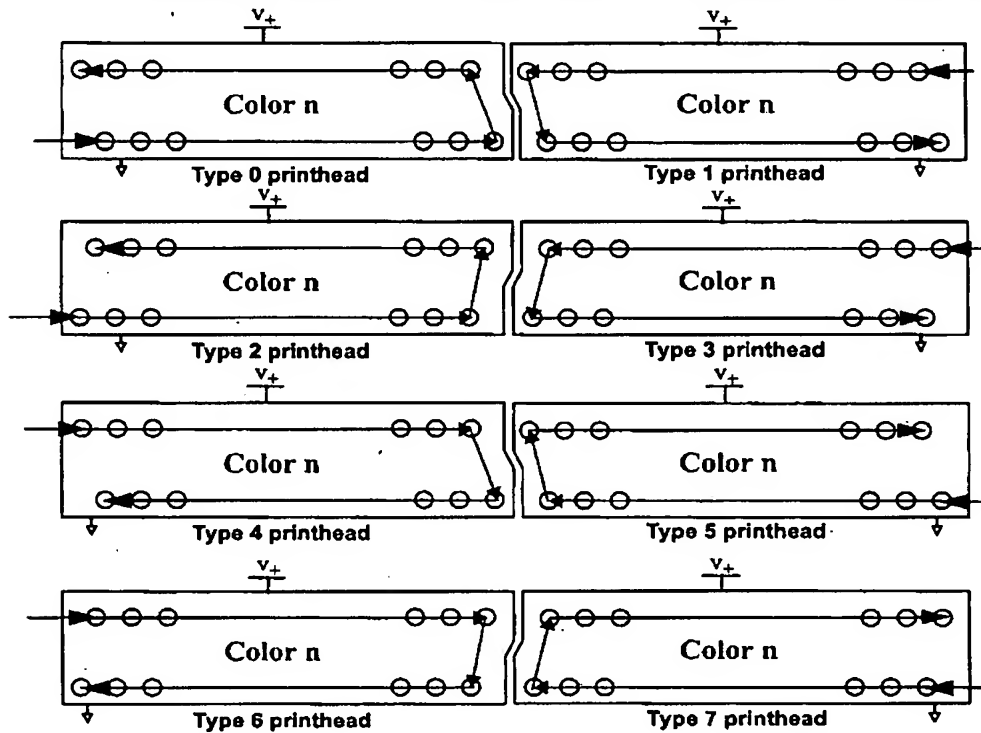

Figure 253. Printhead Types 0 to 7

Table 167 defines the printhead pairing and location of the each printhead type, with respect to the flow of paper, for the 8 possible arrangements

Table 167. Definition of the different printhead arrangements

Printhead Arrangement	Printhead on left side, with respect to the flow of paper	Printhead on right side, with respect to the flow of paper
Arrangement 1	Type 0	Type 1
Arrangement 2	Type 1	Type 0
Arrangement 3	Type 2	Type 3
Arrangement 4	Type 3	Type 2
Arrangement 5	Type 4	Type 5
Arrangement 6	Type 5	Type 4
Arrangement 7	Type 6	Type 7
Arrangement 8	Type 7	Type 6

35.4   BILITHIC PRINTHEAD SYSTEMS

When using the bilithic printheads, the position of the power/gnd bars coupled with the physical footprint of the printheads mean that we must use a specific pairing of printheads together for printing on the same side of an A4 (or wider) page, e.g. we must always use a Type 0 printhead with a Type 1 printhead etc.

While a given printing system can use any one of the eight possible arrangements of printheads, this document only presents two of them, Arrangement 1 and Arrangement 2, for purposes of illustration. These two arrangements are discussed in subsequent sections of this document. However, the other 6 possibilities also need to be considered.

The main difference between the two printhead arrangements discussed in this document is the direction of the paper flow. Because of this, the dot data has to be loaded differently in Arrangement 1 compared to Arrangement 2, in order to render the page correctly.

35.4.1   Example 1: Printhead Arrangement 1

Figure 254 shows an Arrangement 1 printing setup, where the bilithic printheads are arranged as follows:
- The Type 0 printhead is on the left with respect to the direction of the paper flow.
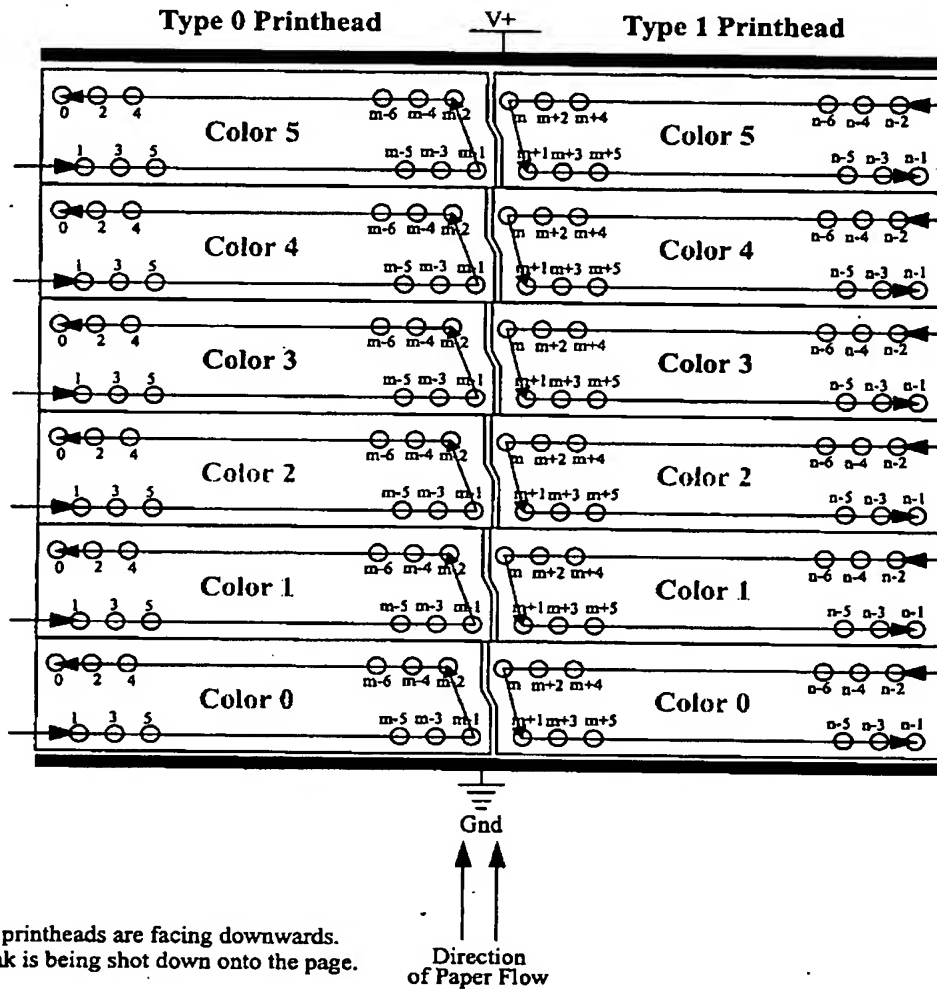- The Type 1 printhead is on the right.

Figure 254. Identification of printheads nozzles and shift-register sequences for printheads in Arrangement 1

Table 168 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 168. Order in which the even and odd dots are loaded for printhead Arrangement 1

Dot Sense	Type 0 printhead when on the left	Type 1 printhead when on the right
Odd	Loaded second in descending order.	Loaded first in descending order.
Even	Loaded first in ascending order.	Loaded second in ascending order.

Figure 255 shows how the dot data is demultiplexed within the printheads.
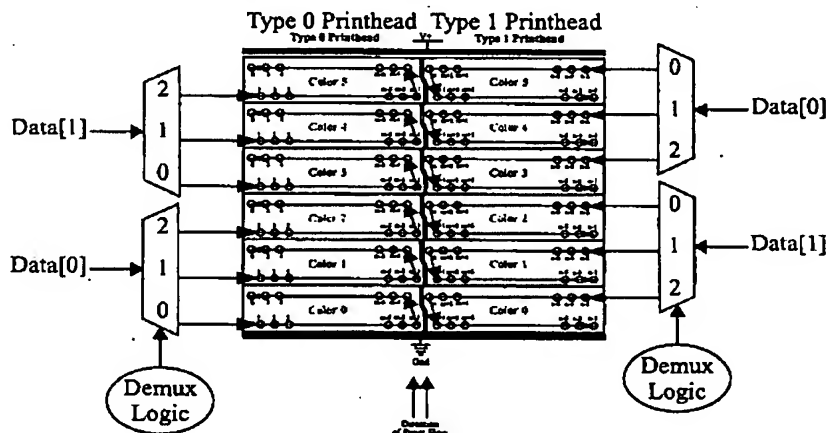

Figure 255. Demultiplexing of data within the printheads in Arrangement 1

Figure 256 and Figure 257 show the way in which the dot data needs to be loaded into the printheads in Arrangement 1, to ensure that color 0-dot 0 appears on the left side of the printed page.
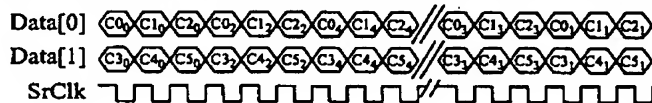

Figure 256. Signalling for a Type 0 printhead in Arrangement 1

Figure 257. Signalling for a Type 1 printhead in Arrangement 1

35.4.2   Example 2: Printhead Arrangement 2

Figure 258 shows an Arrangement 2 printing setup, where the bilithic printheads are arranged as follows:
*   The Type 1 printhead is on the left with respect to the direction of the paper flow.
*   The Type 0 printhead is on the right.

The printheads are facing downwards.
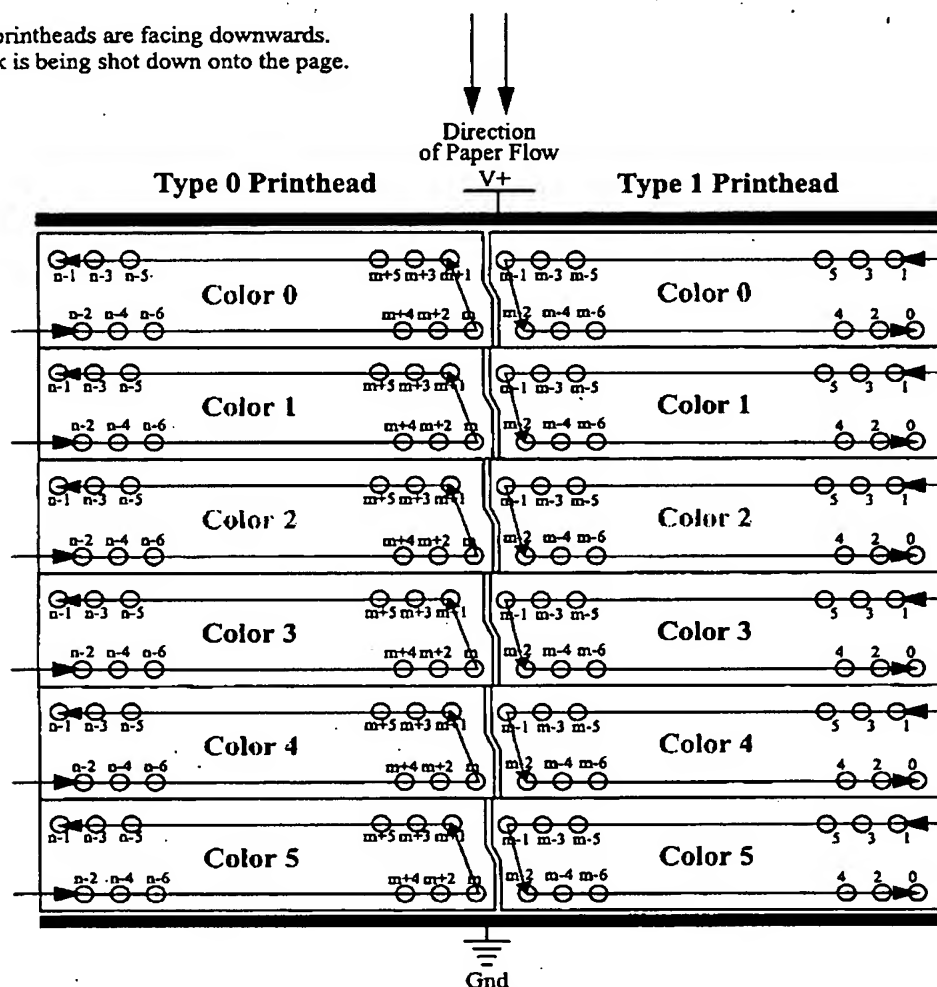The ink is being shot down onto the page.

Direction
of Paper Flow
V+

Type 0 Printhead   **Type 1 Printhead**

Figure 258. Identification of printheads nozzles and shift-register sequences for printheads in Arrangement 2

Table 169 lists the order in which the dot data needs to be loaded into the above printhead system, to ensure color 0-dot 0 appears on the left side of the printed page.

Table 169. Order in which the even and odd dots are loaded for printhead Arrangement 2

Dot Sense	Type 0 printhead when on the right	Type 1 printhead when on the left
Odd	Loaded first in descending order.	Loaded second in descending order.
Even	Loaded second in ascending order.	Loaded first in ascending order.

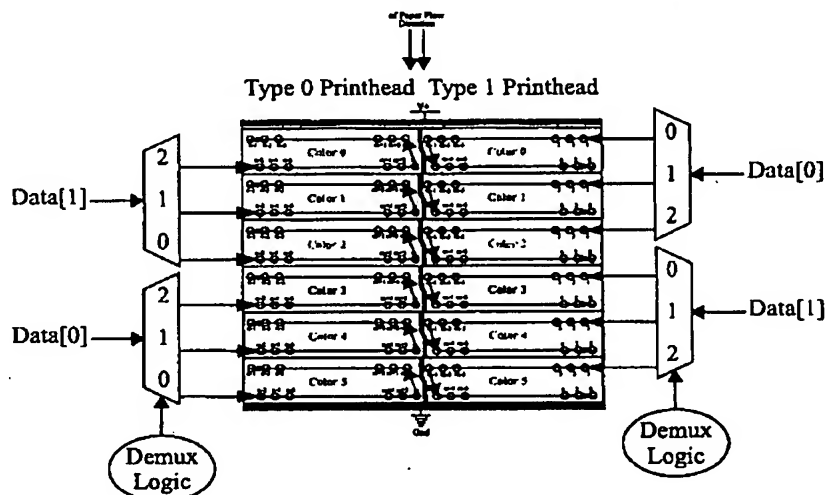Figure 259 shows how the dot data is demultiplexed within the printheads.

Figure 259. Demultiplexing of data within the printheads in Arrangement 2

Figure 260 and Figure 261 show the way in which the dot data needs to be loaded into the printheads in Arrangement 2, to ensure that color 0-dot 0 appears on the left side of the printed page.
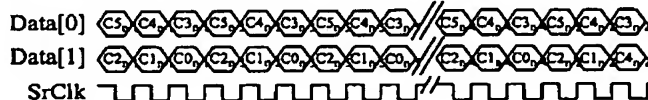

Figure 260. Signalling for a Type 0 printhead in Arrangement 2

Figure 261. Signalling for a Type 1 printhead in Arrangement 2

35.4.3   Conclusions

Comparing the signalling diagrams for Arrangement 1 with those shown for Arrangement 2, it can be seen that the color/dot sequence output for a printhead type in Arrangement 1 is the reverse of the sequence for same printhead in Arrangement 2 in terms of the order in which the color plane data is output, as well as whether even or odd data is output first. However, the order within a color plane remains the same, i.e. odd descending, even ascending.

From Figure 262 and Table 170, it can be seen that the plane which has to be loaded first (i.e. even or odd) depends on the arrangement. Also, the order in which the dots have to be loaded (e.g. even ascending or descending etc.) is dependent on the arrangement.

If the device controlling the printheads can re-order the bits according to the following criteria, then it should be able to operate in all the possible printhead arrangements:

- Be able to output the even or odd plane first.
- Be able to output even and odd planes in either ascending or descending order, independently.
- Be able to reverse the sequence in which the color planes of a single dot are output to the printhead.
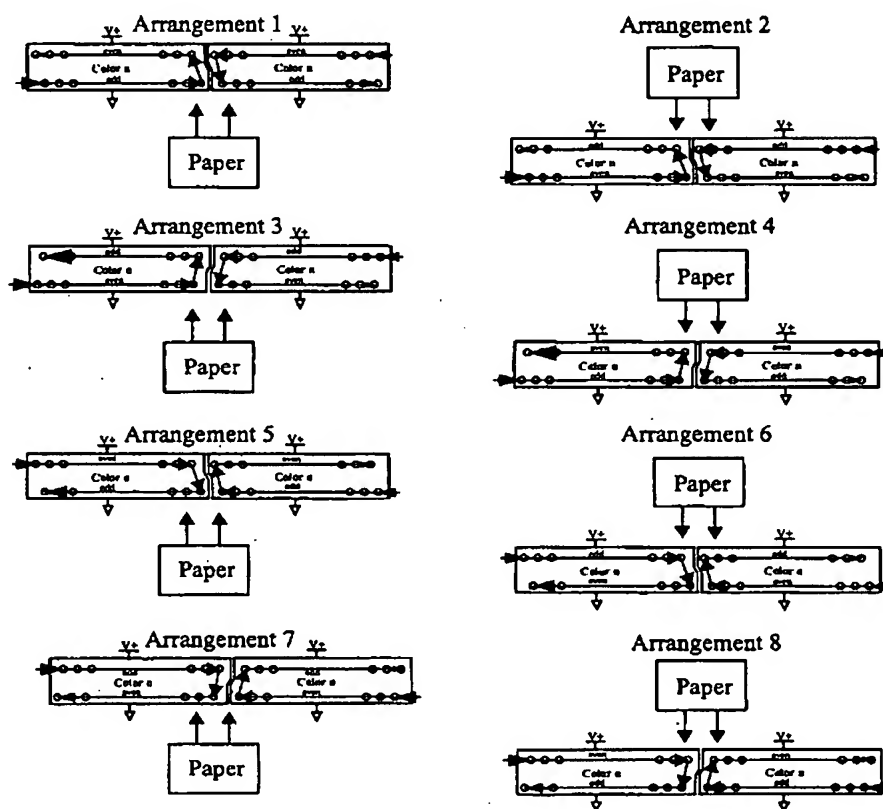

Figure 262. All 8 Printhead Arrangements

Table 170. Order in which even and odd dots and planes are loaded into the various printhead arrangements

Printhead Arrangement	Left side of printed page	Right side of printed page
Arrangement 1	Even ascending loaded first	Odd descending loaded first
	Odd descending loaded second	Even ascending loaded second
Arrangement 2	Odd descending loaded first	Even ascending loaded first
	Even ascending loaded second	Odd descending loaded second

SoPEC : Hardware Design

Table 170. Order in which even and odd dots and planes are loaded into the various printhead arrangements

Printhead Arrangement	Left side of printed page	Right side of printed page
Arrangement 3	Odd ascending loaded first Even descending loaded second	Even descending loaded first Odd ascending loaded second
Arrangement 4	Even descending loaded first Odd ascending loaded second	Odd ascending loaded first Even descending loaded second
Arrangement 5	Odd ascending loaded first Even descending loaded second	Even descending loaded first Odd ascending loaded second
Arrangement 6	Even descending loaded first Odd ascending loaded second	Odd ascending loaded first Even descending loaded second
Arrangement 7	Even ascending loaded first Odd descending loaded second	Odd descending loaded first Even ascending loaded second
Arrangement 8	Odd descending loaded first Even ascending loaded second	Even ascending loaded first Odd descending loaded second

REFERENCES

36 Silverbrook References

[1]   Silverbrook Research, *PEC ASIC Hardware Design Version 2.5, 26 April 2002*
[2]   Silverbrook Research, *Bi-lithic Printhead Specification, Released 7 June 2002*
[3]   Silverbrook Research, *Software Design Specification (Draft), Version 0.3, 19 October 2001*
[4]   Silverbrook Research, *SoPEC Stepper Motor Outputs, Released 23 May 2002*
[5]   Silverbrook Research, *Netpage System Overview, 2000.*
[6]   Silverbrook Research, *Authentication Chip, 1998*
[7]   Silverbrook Research, *Authentication of Consumables, 1998.*
[8]   Silverbrook Research, *QA Chip Interface Description Version3.5 Draft, 19 July 2002.*
[9]   Silverbrook Research, *SoPEC Security Overview, Version1.3, August 2002.*
[10] Silverbrook Research, *SoPEC/MoPEC Bilithic Printhead Reference, Version 1.0 Draft, 21 October 2002.*

37 S3 References

[11] Silicon Software Systems, *SoPEC Requirement Specification, Version 1.1 Released 13 June 2002*
[12] Silicon Software Systems, *PEC ASIC: SDRAM Controller Interface Unit Specification, 2001*
[13] Silicon Software Systems, *Print Engine Controller, DataBook, version 1.1, Released 13 May 2002*
[14] Silicon Software Systems, *Error analysis of color conversion options for PEC, rev2.0, April 2001.*
[15] Silicon Software Systems, *SoPEC Software Debug , Version 0.1, August 2002.*

38 ASIC Vendor References

[16] IBM Cu-11 Databook: Macros, March 21 2002.

39 Other References

[17] Amphion, 2001, *CS6150 Motion JPEG Decoder Databook*, Amphion Semiconductor Ltd.
[18] ANSI/EIA 538-1988, *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Equipment*, August 1988
[19] Bender, W., P. Chesnais, S. Elo, A. Shaw, and M. Shaw, *Enriching communities: Harbingers of news in the future*, IBM Systems Journal, Vol.35, Nos.3&4, 1996, pp.369-380
[20] CCIR Rec. 601-2, *Encoding Parameters of Digital Television for Studios*, Recommendations of the CCIR, 1990, Vol XI-Part 1, Broadcasting Service (Television), pp.95-104.
[21] Farrell, J., *How to Allocate Bits to Optimize Photographic Image Quality*, Proceedings of IS&T International Conference on Digital Printing Technologies, 1998, pp.572-576
[22] Humphreys, G.W., and V. Bruce, *Visual Cognition*, Lawrence Erlbaum Associates, 1989, p.15
[23] ISO/IEC 19018-1:1994, *Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines*, 1994
[24] Lyppens, H., *Reed-Solomon Error Correction, Dr. Dobb's Journal* Vol.22, No.1, January 1997
[25] Olsen, J. *Smoothing Enlarged Monochrome Images*, in Glassner, A.S. (ed.), *Graphics Gems*, AP Professional, 1990
[26] Rorabaugh, C, *Error Coding Cookbook*, McGraw-Hill 1996
[27] Thompson, H.S., *Multilingual Corpus 1* CD-ROM, European Corpus Initiative
[28] Urban, S.J., *Review of standards for electronic imaging for facsimile systems*, Journal of Electronic Imaging, Vol.1(1), January 1992, pp.5-21
[29] Wallace, G.K., *The JPEG Still Picture Compression Standard*, Communications of the ACM, Vol.34, No.4, April 1991, pp.30-44

[30] Wicker, S., and Bhargava, V., *Reed-Solomon Codes and their Applications*, IEEE Press 1994

[31] Yasuda, Y., *Overview of Digital Facsimile Coding Techniques in Japan*, Proceedings of the IEEE, Vol. 68(7), July 1980, pp.830-845

[32] SPARC International, *SPARC Architecture Manual, Version8, Revision SAV080SI9308*

[33] Gaisler Research, *The LEON-2 Processor User's Manual, Version 1.0.7*, September 2002